**KAUNO TECHNOLOGIJOS UNIVERSITETAS**
**INFORMATIKOS FAKULTETAS**
**INFORMACIJOS SISTEMŲ KATEDRA**

**Povilas Šafranauskas**

# Veiklos modelio ir vartotojo reikalavimų (Use-Case) modelio sąsajos tyrimas

**Magistro darbas**

**Vadovas: prof. Saulius Gudas**

**Kaunas, 2007**

**KAUNO TECHNOLOGIJOS UNIVERSITETAS**
**INFORMATIKOS FAKULTETAS**
**INFORMACIJOS SISTEMŲ KATEDRA**

**Povilas Šafranauskas**

# Veiklos modelio ir vartotojo reikalavimų (Use-Case) modelio sąsajos tyrimas

**Magistro darbas**

**Recenzentas**

**doc. S. Maciulevičius**
**2007-05-22**

**Vadovas**

**prof. S. Gudas**
**2007-05-22**

**Atliko**

**IFM -1/1 gr. stud.**
**P. Šafranauskas**
**2007-05-22**

**Kaunas, 2007**

# Turinys

## Lentelių sąrašas



## Paveikslų sąrašas

# Įvadas

Mūsų dienomis informacinės sistemos tampa vis sudėtingesnės ir verslo procesams aprašyti seniai nebeužtenka paprastų tekstų ar schemų. Kompiuterizuotų IS reikalavimams aprašyti ir modeliuoti geriausiai tinka specialiai tam skirti modeliai. Modeliavimas tapo būtinas kuriant dideles, sudėtingas ar su kitomis sistemomis sąveikaujančias informacines sistemas, nors daugelis programuotojų siekia minimizuoti modeliavimo pastangas. Modeliuojami ne tik vartotojo poreikiai, dalykinė sritis, programinė įranga, bet ir organizacijos veikla - pastaraisiais metais ši modeliavimo fazė tapo daugelio projektavimo procesų dalimi. Todėl specialūs kompiuterizuoti modeliavimo įrankiai tapo neatskiriama sudėtingų IS kūrimo ir projektavimo dalimi. Modernėjant technologijoms vis labiau orientuojamasi į reikalavimų specifikavimo fazę IS kūrimo projektuose – reikalavimai turi lemiamą reikšmę projektuojant modernias ir sudėtingas informacines sistemas. Todėl vis didėja projektavimo įrankių įtaka IS kūrimo procese, jų pagalba kartais generuojant ir kodą. Taigi, kad tinkamai būtų sukurta ir vystoma nauja sistema, būtina tinkamai apsibrėžti viziją, veiklos modelį ir tiksliai specifikuoti reikalavimus. Šį procesą, bei veiklos modelio transformavimą į reikalavimų sistemai modelį aptarsiu šiame darbe.

Žemiau pateikiami darbo tikslai ir uždaviniai, išsikeliama problema ir pagrindžiamas idėjos naujumas. Kituose skyriuose pateikiama analogų analizės rezultatai, transformacijos algoritmas, prototipo projektas ir realizacijos rezultatai.

**Darbo tyrimo objektas** yra automatizuotas veiklos procesus aprašančių modelių transformavimo į kitus modelius procesas, grindžiamas veiklos modelio ir vartotojo reikalavimų veiklą automatizuojančiai sistemai, sąsaja.

**Darbo tikslas** yra atrasti verslo procesams, transakcijoms ir reikalavimams aprašyti naudojamų Veiklos ir UCM modelių sąlyčio taškus bei aprašyti sąsają tarp jų. Aprašant UCM modelio atvaizdavimą iš Veiklos modelio bus remiamasi UML 2.0 specifikacija [5].

Darbo **uždaviniai** yra:

- iškelti reikalavimų aprašymo problemą, ją analizuoti ir numatyti sprendimą;
- aprašyti metodiką, kuri bus taikoma verslo procesų poreikių modeliavimui;
- išanalizuoti sistemoje egzistuojančių sprendimų aibę ir galimybes;
- susisteminti ir aprašyti metodiką bei algoritmus, kuriais remiantis bus projektuojamas sprendimas;
- pateikti UCM transformavimo Veiklos modelio pagrindu algoritmą, aprašyti jo žingsnius ir iliustruoti pavyzdžiais;
- suprojektuoti ir paruošti sprendimą realizacijai;

- realizuoti sprendimą pasirinkto įrankio pagalba.

**Mokslinės problemos esmė** modelio transformacijų sukūrimas remiantis loginiais ryšiais ir sąsajomis verslo automatizavimo sprendimams. Šis praktinis sprendimas įrodo stiprų ryšį tarp veiklos modelio ir reikalavimų to proceso automatizavimo sprendimui.

Šiuo metu verslo modeliai kuriami įvairių anotacijų pagalba (BPMN, BPEL, UML), pagal kuriuos vėliau braižomi reikalavimai sistemai, ar generuojamas kodas (modeliu grįsta architektūra). Tuo tarpu naudojant UML Veiklos diagramas, aprašant verslo modelius MDA PIM modeliavimo lygmenyje, automatizuota transformacija į kitus modelius nevykdoma. Todėl modeliuojant reikalavimus aprašytiems ir sumodeliuotiems verslo procesams tenka juos braižyti atskirai neatmetant ir klaidos faktoriaus dėl ne visada pakankamos kvalifikacijos. Šiuo atveju laiko ir korektiškumo problema yra ryškiausia.

**Darbo rezultatų mokslinis naujumas** siejamas su aprašytuoju metodu, leidžiančiu automatizuoti diagramų transformacijas, verslo modeliavimo kontekste. Metodas palaiko transformacijas pagal UML standartus.

Pasiūlytas metodas grindžiamas efektyvesnius sprendimu. Tradiciškai diagramos informacija eksportuojama į XML failą ir pakoregavus ar pakeitus tą failą ir jo atributus importuojama atgal, išgaunant kito tipo diagramas. Remiantis šiuo metodu, vykdoma tiesioginė transformacija vienos diagramos elementus atvaizduojant į atitinkančius kitos diagramos elementus. Ši metodika remiasi Modeliavimu grįsta architektūra ir atitinka PIM modelio metodiką.

**Praktinė darbo svarba.** Darbe suprojektuotas pasiūlytąjį metodą palaikantis ir pagrindžiantis sprendimas, sukurta realizacija UML įrankio aplinkoje. Lyginant su modeliavimu rankiniu būdu, automatizuotas procesas mažina laiko sąnaudas ir klaidos tikimybę, išlaikant diagramų korektiškumą pagal UML notaciją. Gauti rezultatai pateikiami darbe. Sukurtas prototipas gali būti tobulinamas ir plečiamos jo galimybės siekiant daugiau efektyvumo ir naudos.

**Ateities perspektyvos.** Remiantis tuo, jog vystant sudėtingas IS vis daugiau sutelkiama dėmesio į sistemos modeliavimą ir kodo inžineriją, modeliavimo įrankiai tampa vis svarbesni, todėl teisingas procesų modeliavimas tampa vis svarbesnis. Vadinasi, sukurtasis sprendimas ateityje bus dar labiau naudojamas ir labiau išplėstas gali tapti ne vieno modeliavimo įrankio dalimi.

**Darbo struktūra.** Pirmoje darbo dalyje aprašoma verslo procesų modeliavimo metodika ir principai, didžiausią dėmesį kreipiant į reikalavimų specifikavimą verslo sričiai ir konkretiems verslo procesams. Aprašoma reikalavimų svarba šiuolaikinių IS kūrime. Pateikiama UML ir

MDA metodologijos, kuriomis remiantis bus kuriami modeliai. Verslo procesų ir reikalavimų kontekste atskirai aprašomos naudojamos diagramos, jų specifika ir paskirtis.

Šio darbo analizės dalyje pateikiama UML CASE įrankių, generuojančių UML modelių diagramas iš kodo ir atvirkščiai, analizė. Aprašomi keletas populiariausių ir plačiausiai pasaulyje naudojamų modelių. Pateikiami įrankiai buvo pasirenkami ir taip, kad kuo labiau atitiktų planuojamo realizuoti sprendimo koncepciją, kad butų galima geriau įžvelgti esamus privalumus ir trūkumus, kuriuos galėtų ištaisyti realizuojamos pasirinkto įrankio adaptacijos.

Metodinėje dalyje aprašomos galimos naudojamų modelių sąsajos ir atvaizdavimo seka. Pateikiami grafiškai ir aprašomi galimi modeliavimo variantai, iš kurių vienas pasirinktas bus realizuotas sekančiame darbo etape. Taip pat pateikiamas Panaudojimo atvejų modelio generavimo iš Veiklos modelio algoritmo aprašymas ir diagrama. Taip pat pateikiami modelių objektų atvaizdavimo vienas į kitą atitinkami elementai.

Ekspertinėje (projektinėje) dalyje pateikiamas praktinės realizacijos, pagrindžiančios UCM modelio generavimą Veiklos modelio pagrindu, projektas ir architektūra. Pateikiami pavyzdžiai ir grafinė vartotojo sąsaja.

Darbo pabaigoje pateikiamos darbo išvados, literatūros sąrašas, terminų ir santrumpų žodynėlis bei priedai.

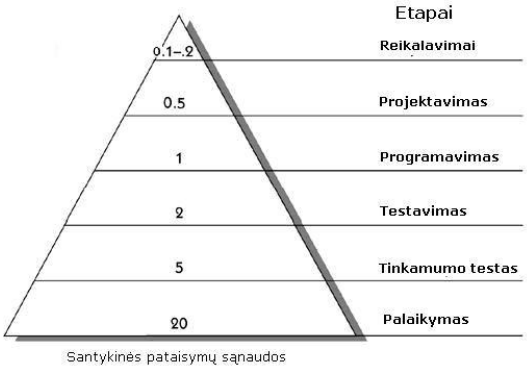
# 1. Verslo procesų reikalavimų modeliavimas

## 1.1 Reikalavimų svarba IS kūrimo procese

Mūsų laikais informacinės sistemos nebėra pagalbinė verslo priemonė – jos jau seniai tapo neatsiejama daugumos verslo organizacijų integruota dalimi, neatsiejamai dalyvaujančia verslo procesuose. Tačiau IS vystymo tempai, apimtys tik nesustojamai toliau didėja. Ypač daug dėmesio skiriama Internetui orientuotų sistemų kūrimui bei įvairių informacinių sistemų integracijai. Augant kuriamos programinės įrangos apimtims ir sudėtingumui, projektavimas tampa vis svarbesne IS inžinerijos dalimi. IS kūrimo ir vystymo projektams tampant sudėtingesniems ir jų apimtims augant, nemažai projektų užbaigiami su viršytomis laiko ar biudžeto sąnaudomis arba nebaigiami išvis.

Todėl IS projektavime, reikalavimų valdymas yra kritinis procesas, galintis užtikrinti sėkmingą viso IS projekto vykdymą ir užbaigimą. Netinkamas reikalavimų valdymas yra labiausiai pasitaikančios IS kūrimo klaidos, kurios kainuoja brangiausiai [1]. Pateiktame 1 pav. matome, kad klaidų pataisymas reikalavimų apibrėžimo stadijoje užima 5-10 kartų mažiau laiko negu jas taisant programavimo metu ir 100-200 kartų mažiau laiko, negu palaikant jau įdiegtą sistemą:

*Šaltinis: Davis, Alan M. Software Requirements: Objects, Functions, and States. Englewood Cliffs, NJ: Prentice-Hall.*




**1. pav. Pataisymų kaštai IS kūrimo stadijose.**

Programinės įrangos kūrimas visada prasideda nuo reikalavimų analizės ir aprašymo. Daugiausia problemų, kuriant programinę įrangą, kyla būtent iš prastai dokumentuotų, nepilnai aprašytų, skirtingai suprantamų bei projekto eigoje besikeičiančių reikalavimų. Labai svarbu suprasti, kad IS kūrimo projektų reikalavimai turi būti nuosekliai analizuojami keliuose lygiuose – nuo **vizijos** ir **verslo tikslų** iki **galutinio vartotojo scenarijų** bei **realizacijos funkcijų aprašymo**.

*Šaltinis: Alistair Cockburn. Writing effective Use Cases. Addison Wesley, 2000.*


**2. pav. Cockburn reikalavimų laivo modelis [7]**

Reikalavimų dokumentavimui paprastai yra paruošiami dokumentų šablonai bei jų pildymo taisyklės ir pavyzdžiai. Siekiant reikalavimų aiškumo, dažnai naudojamos grafinės UML diagramos:

- klasių diagramos – verslo srities objektų ir jų ryšių modeliavimui;
- panaudojimo atvejų diagramos (UCM) – verslo procesų ir funkcinių reikalavimų realizacijos vizualizavimui;
- veiklos diagramos – vartotojo scenarijų, panaudos atvejų realizacijos vizualizavimui;

*Iš* Veiklos (*Activity)* modelio patogu atvaizduoti ir išplėstą Komponentinį modelį, kuris papildo verslo procesų logiką.

Programinės įrangos rinkoje siūloma daug UML įrankių, leidžiančių patogiai ir efektyviai modeliuoti projektavimo sprendimus ir apibrėžti reikalavimus. Dauguma UML CASE įrankių turi galimybes generuoti kodą iš UML diagramų, naudojant kodo inžineriją. Plačiausiai naudojamas generavimas iš Klasių diagramų. Kiek mažiau paplitusi kodo inžinerija iš Veiklos diagramų. Dar naudojama Sekų diagramų bei kitų mažai naudojamų schemų generavimas į kodą. Nei vienas rinkoje esantis analizuotas sprendimas neturi galimybės generavime panaudoti Komponentinį modelį, kuris puikiai papildo Veiklos modelį verslo procesų modeliavime. Veiklos modelis, aprašantis veiklos procesus ir duomenų srautus, paprastai yra pagrindas vėliau kuriamiems UCM modeliams, kurie apibrėžia kompiuterizuojamus uždavinius vartotojo reikalavimams. Todėl UCM atvaizdavimas iš Veiklos modelio, panaudojant Komponentinį modelį būtų naudingas aprašant projektuojamos IS elgseną ir reikalavimus.

## 1.2 UML taikymas projektuojant IS

Šiuo metu programinės įrangos industrijoje yra plačiai naudojama UML (Unified Modeling Language) modeliavimo kalba, leidžianti aprašyti projektavimo sprendimus. UML yra vizuali kalba, apibrėžianti grafinę anotaciją, skirtą įvairių programinės įrangos architektūros aspektų modeliavimui. Sakoma, kad paveikslėlis vertas tūkstančio žodžių, o UML modelis dar daugiau. UML modeliai taip pat dar vadinami programinės įrangos žemėlapiais — jie leidžia greičiau ir lengviau suprasti programinės įrangos struktūrą ir veikimo principus, todėl yra efektyviai panaudojami programinės įrangos architektūros dokumentavimui bei projektavimo sprendimų aptarimui.

Šiuo metu UML modeliavimo kalba yra standartinė projektavimo priemonė, kuri visuotinai naudojama pasaulyje ir praktiškai neturi rimtesnių "konkurentų". Todėl, jeigu reikia standartizuoti modeliavimo anotaciją, kuri būtų plačiai suprantama, vienareikšmiškai rekomenduojama pasirinkti UML. Šiuo metu esanti naujausia ir aktuali versija, UML 2.0 apibrėžia 13 rūšių diagramas, kurios leidžia specifikuoti įvairius architektūros aspektus. Tačiau tikrai nebūtina naudoti visų diagramų. Čia, kaip ir daugelyje kitų sričių galioja 80/20 taisyklė: 80% projekto užtenka 20% UML galimybių.

Informacinių sistemų projektavimo metodai nurodo sistemos projektavimo veiksmų seką – kaip, kokia tvarka ir kokias UML diagramas naudoti projektavimo procese bei kaip tą procesą vykdyti. Daugelis jų remiasi kelių tipų diagramomis, aprašančiomis skirtingų sistemos aspektų savybes. Kiekvienos diagramos prasmę galima nusakyti atskirai, bet svarbiau yra tai, kad kiekviena diagrama yra visos sistemos modelio projekcija.

Vis dėlto toks sistemos aprašymas yra gana painus, nes daugumoje diagramų pateikiama informacija iš dalies sutampa ir aprašo tuos pačius dalykus tik skirtingais aspektais. Be to, nepatyrusio specialisto netinkamai naudojamos UML diagramos gali būti nenuoseklios, nepilnos ir nevienareikšmiškai aprašyti sistemą. Dabartiniai UML CASE įrankiai tokiais atvejais labai nedaug kuo tegali padėti projektuotojui. Taigi automatizuoto projektavimo įrankiai labai palengvina projektuotojo darbą, tačiau vis dėlto nepakankamai jį automatizuoja. Didelis dėmesys skiriamas modeliavimui, ir projektuotojas šiai veiklai turi sugaišti nemažai laiko, kai galutinis darbo rezultatas vis dėlto yra programa, o ne modeliai. Yra siekiama kiek įmanoma formalizuoti modelių sudarymą, tam sukurta įvairių metodikų, tačiau dar egzistuoja ir neišnaudotų galimybių, kurios leistų patobulinti CASE įrankiais valdomus informacinių sistemų projektavimo metodus. Formalizavimas atveria naujas programinės įrangos ir informacinių sistemų projektavimo, kūrimo galimybes.

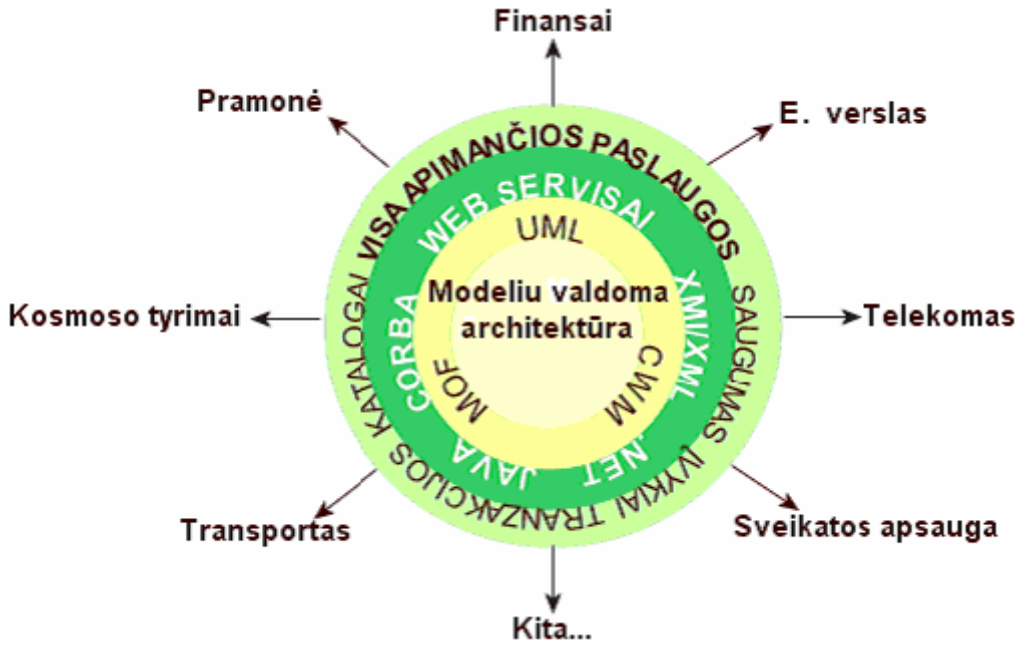
Pasiekus maksimalų UML modelių suderinamumą, kai modeliai tarpusavyje tvirtai susiję pagal aiškiai nusakytas taisykles, griežčiau išreikšti ir pilnesni, labai palengvėtų automatizuoto programinės įrangos kūrimo uždavinio sprendimas.

Aišku, nebūtinai reikia specializuotų UML įrankių – nedideliuose projektuose galima braižyti UML diagramas bendrais diagramų modeliavimo įrankiais, pvz. Microsoft Visio, arba tiesiog ranka ant popieriaus ar lentos. Visgi didesniuose projektuose specializuotų UML įrankių naudojimas leidžia dirbti daug efektyviau.

Nuo tada (1997 gruodis), kai OMG (Object Management Group) paskelbė UML kaip standartą, ši vis labiau populiarėjanti kalba daro didžiulį poveikį IS (informacinių sistemų) projektavimui. Pirmaisiais šios kalbos naudojimo metais, ji buvo naudojama modeliuoti informacinėms sistemoms, tačiau ši kalba yra labai tinkama modeliuoti ir verslo procesams – ji spėjo paplisti ir verslo analitikų tarpe. UML tinkama apibrėžti verslo struktūrinius, elgsenos ir taisyklių aspektus. Taigi, šios kalbos naudojimas IS ir verslo modeliavime gali užtikrinti dokumentacijos stabilumą, vienodumą ir padėti lengviau susikalbėti IS projektuotojams su verslo kūrėjais [2].

### 1.3 UML formalizavimas panaudojant MDA

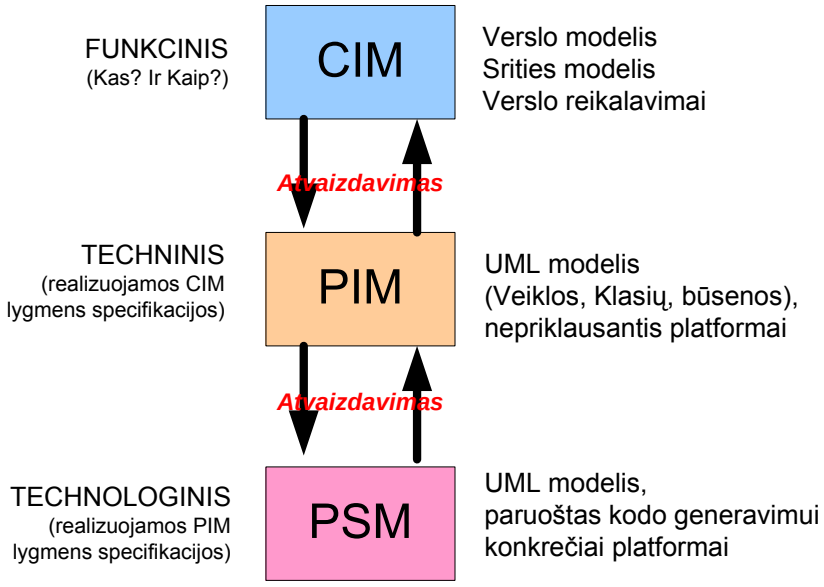
Siekis formalizuoti UML, pasiekti maksimalų modelių suderinamumą yra susijęs su IS kūrimo idėja, pagal kurią siekiama atskirti verslo arba programos logiką nuo technologinės realizacijos. Šis metodas dar vadinamas modeliu grįsta architektūra arba MDA [4] (*Model Driven Architecture*) ir laikomas ateities programinės įrangos kūrimo pagrindu. MDA apibrėžia būdą, kaip atskirti sistemos funkcionalumo specifikaciją nuo jos įgyvendinimo tam tikroje platformoje. Tai leidžia automatizuoti informacinių sistemų kūrimo procesą pradedant nuo reikalavimų surinkimo, vėliau juos registruojant modeliuose bei iš jų generuojant būsimos sistemos struktūrą, kuri gali būti realizuota įvairiose platformose. Šio metodo sėkmę iš dalies lemia galimybė generuoti sistemos struktūrą turint UML modelius, taigi šiam uždaviniui reikalingas pilnas UML modelių suderinamumas. MDA aspektu, UML kalba labiau laikoma programavimo kalba, negu modeliavimo, kaip įprasta taikyti UML. Programavimas modeliavimo kalbų pagalba gali ženkliai pagerinti kuriamų IS kokybę, našumą ir ilgaamžiškumą[9] .




**3. pav. MDA struktūra ir panaudojimo sritys**

Pagal MDA, programinės įrangos kūrimo metu sudaromus UML modelius galima suskirstyti į tris grupes:

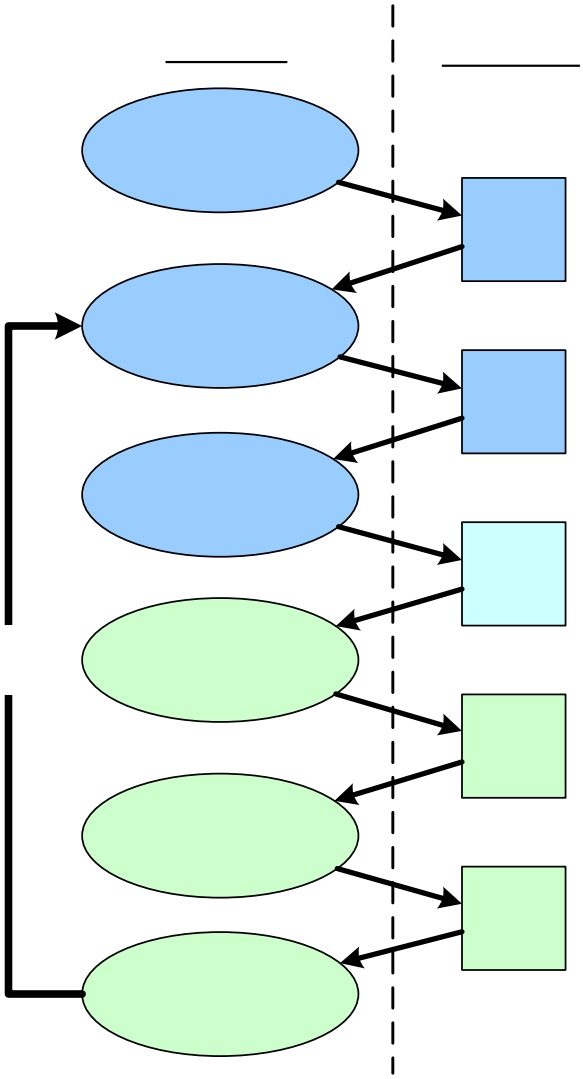
- **CIM** (Computation Independent Model). Tai neformalus modelis, kuris apibūdina modeliuojamos sistemos aplinką bei jai keliamus funkcinius reikalavimus. Šis modelis programos kodo generavimo procese nenaudojamas.
- **PIM** (Platform Independent Model). Tai formalus modelis, kuriame sistema apibūdinama nepriklausomai nuo realizacijos technologijos. Šiame modelyje apibrėžiama sistemos architektūra, loginė struktūra, komponentai bei jų sąveika manipuliacijų duomenimis lygyje.
- **PSM** (Platform Specific Model). Tai PIM modelis, papildytas arba transformuotas taip, kad jame būtų atspindima tam tikrai realizacijos platformai arba jų aibei būdinga informacija.




**4. pav. CIM, PIM ir PSM modelių sąsaja**

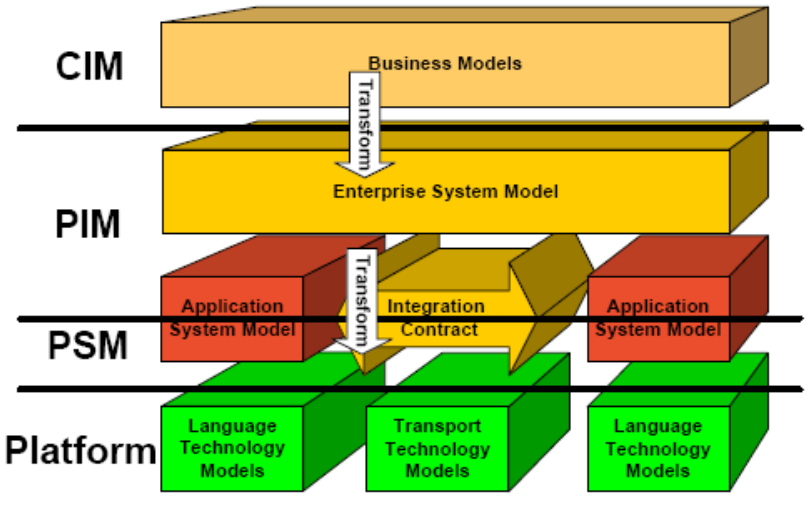
PIM modelis yra atvaizduojamas iš CIM modelio – modelio, kuris atvaizduoja verslo srities sąvokas, kurios nėra visiškai susiję su kompiuterizavimu. Jei verslo procesai yra aprašyti tokiu modeliu, tada pakankamai paprasta pradėti objektiškai orientuotą projektavimą. Tačiau labai dažnai naudojami bendriniai (archetipiniai) verslo (biznio) modeliai, taikliai atvaizduojantys reikiamą verslo sritį, kurių pagalba sugeneruojamas PIM modelis. Kadangi PIM apibrėžia tik būsimos sistemos logiką, sekančiame etape PIM logika turi būti įgyvendinta viename ar keliuose specifiniuose platformų modeliuose PSM . PSM yra išeities (kodo) tekstai arba jų grafinis vaizdas išreikštas UML diagramomis, orientuotomis į platformą.

Kad geriau įsivaizduotume proceso prasme MDA gristą projektavimą, pateikiama kita schema, kurioje pavaizduotas MDA gyvavimo ciklas – nuo reikalavimų analizės iki realizacijos:



**5. pav. MDA paremtos IS vystymo gyvavimo ciklas**

MDA metodologijos pagrindu, remiantis verslo, organizacijos ir IS sąvokomis sistemų architektūroje, pateikiama dar viena schema, atvaizduojanti modelių seką ir vietą Biznio modelio atvaizdavimo ciklo iki realaus kodo požiūriu [11].




**6. pav. MDA meta-modelis**

Schemoje puikiai matosi, kad CIM modelis atvaizduoja Verslo modelius ir pateikia sąvokas, kurios gali egzistuoti be jokios IS. PIM ir PSM atvaizduoja Organizacijos ir Sistemos modelius.

1. Lentelė. MDA meta-modelio lygmenys

| Požiūris / Modelis | **Loginis (informacinis) požiūris** | **Elgsenos požiūris** |
|---|---|---|
| **CIM** (Verslas) | Biznio srities modelis (Ontologija) | Biznio procesų modelis |
| **PIM** (Organizacija ir IS) | Atvaizdavimo šablonai | Veiklos - WFM, Activity. Komponentinis modelis |
| **PSM** (Organizacija ir IS) | Atvaizdavimo schemos (pvz., XML) | Komponentų sąsajos ir metodai |

CIM modelis loginiu požiūriu apibrėžia sąvokas ir ryšius Verslo procese. Šioje vietoje galima panaudoti UML profilį, kaip viduriniojo lygmens ontologiją. Pvz., PIM modelio stereotipai galėtų būti Esybės, Rolės ir Veiklos. Šie stereotipai gali pasitarnauti Biznio srities modelį vaizduojant Komponentiniu modeliu. Veiklos (Biznio procesų) modelis turi Veiklų ir Duomenų srautų stereotipus, kurie gali būti lengvai atvaizduojami projektuojamos organizacijos IS Veiklos ir UCM modeliais. UCM modelio veikėjai (actors) turi sąryšį su organizacijos įvykiais veiklos kontekste – būsimais panaudojimo atvejais. Kuo geriau atvaizduojami veikėjų sąryšiai su Sistemos ribomis – kompiuterizuojamais uždaviniais, tuo glaudesnis ryšys atsiranda su Komponentiniu modeliu ir jo sąsajomis.

Taigi matome, kad šiame darbe orientuojamasi į projektavimo etapą prieš PIM sudarymą – reikalavimų apibrėžimą, kuris turi apibrėžti veiklos procesus ir vartotojo reikalavimus būsimai sistemai. Vadinasi, reikia specifikuoti verslo modelio veiklos funkcijas ir būsenas, sudarant į CIM orientuotą Veiklos modelį. Juo remiantis bus galima sudaryti logiką vaizduojantį PIM modelį. Tam naudojami UML 2.0 [5] V*eiklos* ir UCM modeliai, bei juos papildančios Komponentų ir Išbaigtumo diagramos. Šie modeliai aprašomi atskirai.

## 1.4 Veiklos modelis

Veiklos diagrama dažniausiai atvaizduoja sistemos funkcinį vaizdą, nes paprastai specifikuoja loginius procesus ar veiklos (biznio) funkcijas. Kiekvienas procesas aprašomas kaip veiksmų ar sprendimų seka, su aiškiu apibrėžimu kada ir kokiomis sąlygomis jie bus atlikti. Pirmiausiai reikia gerai suvokti veiklos sritį ir jos procesus tam, kad tinkamai aprašyti būsimos IS elgseną [3]. Veiklos diagramos leidžia parinkti seką, pagal kurią bus vykdomi veiksmai, kitaip sakant, modelis nustato veiklos taisykles, kuriomis reikia vadovautis atliekant veiksmus. Tai yra itin svarbu verslo procesų modeliavime, kadangi procesai labai dažnai vyksta lygiagrečiai [6].

Taigi Veiklos diagramos didžiausia nauda ir pranašumas – biznio procesų ir elgsenos modeliavimas.

Veiklos diagrama puikiai modeliuoja lygiagrečius procesus ir elgseną. Tai labai naudinga specifikuojant darbų sekas biznio procesuose. Veiklos diagramos dažnai padeda skaitytojui ar naudotojui geriau suvokti konkrečius panaudojimo atvejus ar veiklas versle. Šio modelio pagrindu patogu identifikuoti sritis, kurios veiklos procesai gali būti patobulinti ar pakeisti efektyvesniam panaudojimui.

Šiame darbe Veiklos diagrama taip pat naudojama apibrėžti UCM modelio generavimo algoritmui (veiksmų sekai) iš Veiklos modelio.

## 1.5 Komponentinis modelis

Daug diskutuojama kuo skiriasi komponentai nuo įprastų klasių. Visgi, komponentai nėra technologija. Žinoma, tai sunku suprasti techninį išsilavinimą turintiems ir analitinį mąstymą išlavinusiems žmonėms. Visgi Komponentinis modelis vaizduoja, kaip naudotojas nori bendrauti su sistema ir kokius ryšius turėti. Naudotojas paprastai nori įsigyti sistemą gabaliukais, moduliais. Ir jie nori, kad naujieji komponentai būtų integruojami su esama IS ir veiktų vieningai su ja. Tai yra labai realus ir logiškas poreikis mūsų laikais. Gaila, kad jis vis dar sunkiai įgyvendinamas [6].

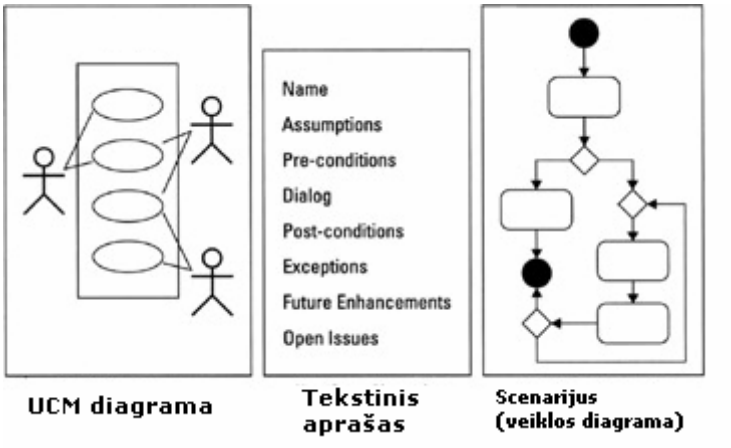
Komponentinis modelis naudojamas tada, kai IS struktūra sudaroma iš komponentų ir sąsajų tarp jų. Kaip UCM modelis atvaizduoja loginę IS architektūrą, Komponentinis modelis apibrėžia fizinį architektūros įgyvendinimą, remiantis apibrėžtu loginiu modeliu. Komponentinis modelis apibrėžia sistemos modulius ir jų tarpusavio ryšius ir sąveikas. Kai kuriais atvejais tas sąveikas tarp komponentų gali nusakyti ir Veiklos diagramos.

Veiklos procesų modeliavimo prasme, komponentai vaizduoja verslo procesus (veiklos modelio atitikmuo yra Veikla (activity) ) ir tų procesų rezultatus arba produktus. Taigi , ryšys su Veiklos modeliu yra aiškus ir apibrėžtas per sistemos procesus.

## 1.6 Vartotojo reikalavimų (panaudojimo atvejų) modelis

Panaudojimo atvejų modelis labiausiai pritaikytas ir populiariai naudojamas funkciniams sistemos reikalavimams aprašyti – diagrama, skirta modeliuoti naudotojų lūkesčius, poreikius kaip turėtų veikti sistema. Diagrama atvaizduoja susijusius su sistema naudotojus, paslaugas ar funkcijas, kurių jei laukia iš sistemos ir veiksmus, kuriuos jie atliks sistemoje [6]. UCM yra tik į panaudojimo atvejus orientuoto požiūrio dalis. Požiūris apima ne tik modelio sudarymą, tačiau ir atskirų panaudojimo atvejų tekstinį aprašymą ir scenarijų sudarymą. Tekstinis aprašymas

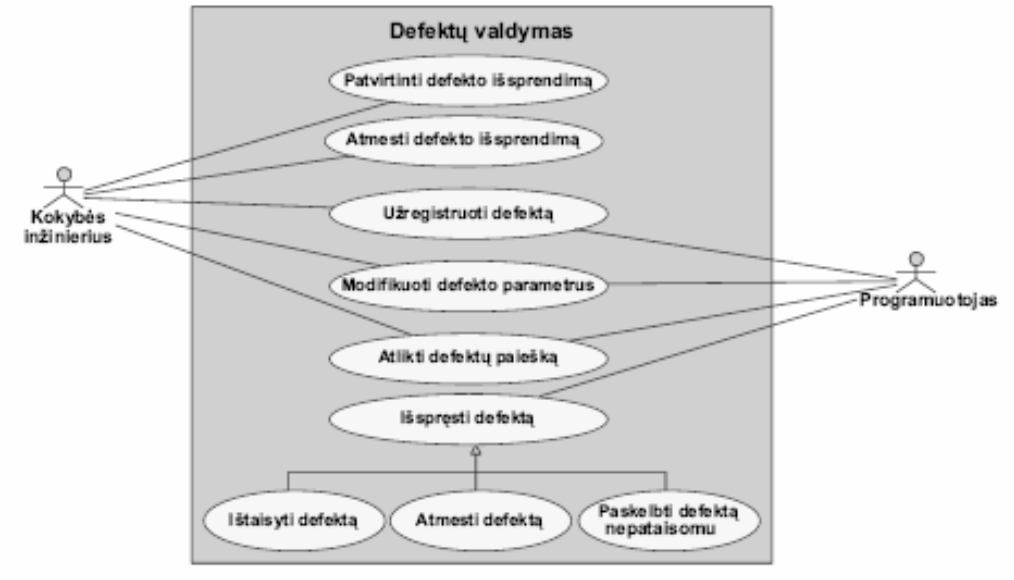
labiausiai atkreipia dėmesį į panaudojimo atvejo detalius reikalavimus. Puikiai matyti, kad panaudojimo atvejai tiksliai atitinka savo pavadinimą pagal paskirtį – tai tam tikra atvaizdavimo forma, aprašanti ką galima panaudoti ar atlikti reikiamoje situacijoje, pvz., verslo procesą, būsimos IS sisteminių reikalavimų aprašymo principus, būsimos IS funkcinius reikalavimus, IS architektūros dokumentaciją [7] . Taigi panaudojimo atvejai gali būti aprašomi UCM modeliu, tekstu arba scenarijais (kuriuos realizuoja Veiklos diagramos):




**7. pav. Panaudojimo atvejų aprašymo galimybės**

Didžiausią dėmesį noriu atkreipti į scenarijų sudarymą, remiantis veiklos modeliu. Tai yra teorinė prielaida, kad UCM modelis **gali būti** generuojamas iš Veiklos modelio. Kai kuriuose situacijose panaudojimo atvejis yra pakankamai paprastas, kad būtų aprašytas tik tekstu. Tačiau mūsų laikais, esant tokiems sudėtingiems veiklos procesams versle, reikalavimų specifikavimas turi būti itin kruopščiai pateiktas. Todėl skaityti tekstą ir sekti kiekvieną galimą atvejį yra tikrai sudėtinga. Paveikslėlis ar diagrama beveik visada yra dešimtis kartų efektyvesnė ir taupanti laiką priemonė negu tekstas. Bene paprasčiausias, efektyviausias būdas scenarijų atvaizdavimui ir yra Veiklos modelis. Šio modelio nauda yra ta, kad iš karto matosi, kuriose vietose yra kritinių taškų ar neišbaigtų kelių. Užtenka sekti nubrėžtas linijas identifikuojant kiekvieną pavaizduotą kelią – atskirą panaudojimo atvejį [3]. Tekstinis aprašymas gali puikiai papildyti jau sumodeliuotą atvaizdą.

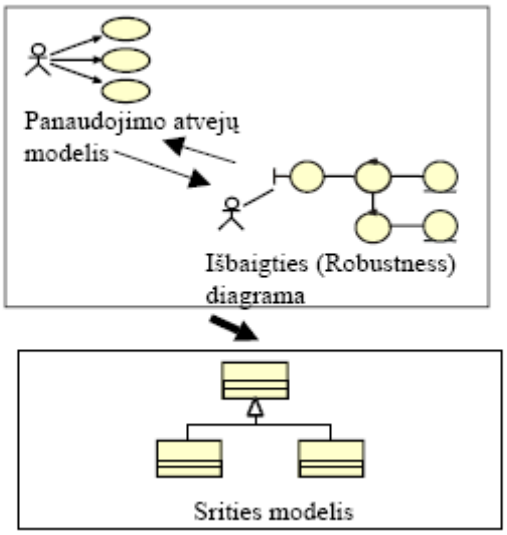
Apie reikalavimų analizės kokybę dažnai galima iš vėliau parengtų vartotojo instrukcijų. Jeigu aprašinėjami vien  vartotojo sąsajos langai ir mygtukai, tačiau neaišku, kaip atlikti veiksmų seką, leidžiančią pasiekti vartotojui reikalingą rezultatą, tai yra indikatorius, kad reikalavimų analizė buvo atlikta tiktai realizacijos lygmenyje, neatsižvelgiant į vartojimo scenarijus. Geriausiai IS vystymo ir kūrimo projektuose galutinio vartotojo lygmenyse reikalavimų analizė atliekama taikant panaudojimo atvejų metodą UCM. Pateikiamas defektų valdymo sistemos panaudojimo atvejų pavyzdys:




**8. pav. Defektų valdymo sistemos panaudojimo atvejai**

## 1.7 Išbaigtumo (Robustness) diagramos

Išbaigtumo diagramų pagrindinė paskirtis – efektyvi metodika, susiejanti analizės fazėje apibrėžtus reikalavimus „kas“ su projektavimo etapo elementais - „kaip“. Šios metodikos tikslas – visų pirma užtikrinti, ar aprašyti verslo procesų panaudojimo atvejai (use cases) yra korektiški ir išbaigti – ar jie pagal reikalavimus tinkamai aprašo projektuojamos IS elgsenos modelius.




**9. pav. Reikalavimų specifikavimas, naudojant Robustness diagramas.**

Išbaigtumo diagramos apibrėžia tokias sąvokas:

- **Aktoriai** (veikėjai) - jie turi tą pačią reikšmę kaip ir UCM modeliuose naudojami aktoriai. Tai su sistema bendraujantys naudotojai.

- **Ribų objektai** (sąsajos) – tai sąsajos, kurias aktoriai naudoja bendravimui su sistema. Tai gali būti beveik bet kokie PĮ elementai – įvairūs langai, ataskaitos, HTML puslapiai ar grafinės vartotojo sąsajos, kuriomis naudojasi aktoriai.
- **Kontrolės objektai** (procesai) – tai sąsaja tarp ribų ir esybių. Šie elementai įgyvendina skirtingų elementų ir jų sąsajų logiką. Dažniausiai vadinami kontroleriais.
- **Esybė (**sritis**)–** tai elementai, paprastai atvaizduojami iš verslo srities modelio.
- **Panaudojimo atvejai** (use cases) – vieni panaudojimo atvejai kartais remiasi į kitus šalia aprašytus panaudojimo atvejus, todėl Robustness diagramose šis elementas naudojamas gana dažnai.

Sudarant Robustness diagramas, remiamasi Panaudojimo atvejų modelio (UCM) logika, pritaikant tokias euristikas:

- *Kiekvieną vartotojo sąsajos elementą (vartotojo langus, ataskaitas, duomenų įvedimą ir pateikimą) žymėti kaip ribos objektą.* Greta to, naudojant panaudojimo atvejus galima detaliau aprašyti, kaip aktoriai dirba su mygtukais ir atskirais laukais. Bet praktika sako, kad geriau „modeliuoti tik tiek, kiek būtinai reikia“, nesileidžiant į nereikalingą detalumą.
- *Naudoti kontrolerius bendriems modeliuojamo scenarijaus procesams valdyti*. Šie elementai tarsi „sulipdo“ diagramą, suteikia jei loginę prasmę proceso kontekste. Projektavime tokie elementai gali būti įgyvendinti kaip darbų sekų procesai, kurių pagrindu kuriami verslo procesų darbų sekų varikliai.
- *Naudoti po kontrolerį kiekvienai biznio taisyklei.* Biznio taisyklių atvaizdavimas suteikia daugiau aiškumo ko iš teisų reikalaujama iš sistemos ir kokios turi būti sąlygos ar apribojimai. Biznio taisykles aprašo veiklos modelis, atvaizduojant reikalavimus modeliuojamiems verslo procesams.
- *Naudoti esybę kiekvienai biznio sąvokai apibrėžti.* Šie elementai naudotini tik tada, kai planuojama vėliau modeliuoti ir naudoti Konceptualųjį modelį.
- *Naudoti panaudojimo atvejo elementą, kai panaudojimo atvejis aprašomas scenarijuje* – atvejis, kai vienas panaudojimo atvejis apibendrina kitus, arba yra bendresnio panaudojimo atvejo dalis.

*Tačiau modeliuojant Robustness diagramas negalima pamiršti Agile principo – turinys visada svarbesnis už atvaizdavimą. Todėl visada reikia atsižvelgti į verslo sritį ir reikalavimus IS.*

Nubraižius Robustness diagramą labai lengva suprasti kokią realizaciją padaryti, kad įgyvendinti apibrėžtus reikalavimų modelio panaudojimo atvejus. Šios diagramos padeda

pamatyti potencialius ir esminius projektuojamos IS elementus. Ribų objektai padeda sujungti panaudojimo atvejus su vartotojo sąsaja ir reikalavimų analize. Robustness diagramos užpildo tarpą tarp verslo procesų ir IS architektūros, apibrėždamos verslo procesų sąvokas kaip būsimos struktūros elementus.

## 2. UML modeliavimo įrankių analizė

Visi egzistuojantys UML įrankiai panašūs savo galimybe sukurti objektiškai orientuotas diagramas, tekstines specifikacijas ataskaitoms bei generuoti kodą. Tačiau jie skiriasi išplėtimo, palaikomų platformų/operacinių sistemų ir papildomų galimybių, skirtingų metodologijų ir kalbų, į kurias generuojamas kodas, specifika. Kiekvienas iš kodo generavimo funkciją teikiančių įrankių turi integruotą kelių (ar keliolikos) kalbų interpretavimo metodą, kuris atvaizduoja numatytus UML modelio elementus į programinio kodo elementus. Norint naudoti UML įrankį, racionalu nustatyti, kada jį reikia naudoti, kokie įrankio taikymo privalumai, kokie yra esminiai kodo generavimo principai bei į ką reikia atsižvelgti, norint pasirinkti konkretų įrankį.

Viena iš svarbesnių priežasčių, lemianti UML įrankio naudojimą programos kodui generuoti, yra ta, kad kodo projektavimas vyksta aukštesniame lygmenyje. Naudojant įrankį, sudaroma UML diagrama, kurios elementai atitinka kodo elementus. UML įrankis suteikia bendrą formą kodo elementams, jų savybėms ir ryšiams tarp elementų žymėti. Taigi po generavimo proceso visi elementai atvaizduojami nuoseklia struktūra – pasiekiama aukštesnė kodo kokybė, kas ypatingai naudinga, jeigu programą realizuoja – o taip dažniausiai ir būna bet kokios didesnės realizuojamos sistemos atveju – komanda, o ne pavienis asmuo. Tai pat tokiu būdu pasiekiamas didesnis kodo kūrimo efektyvumas – generuojama greičiau, negu kodą rašant ranka. Dar viena priežastis, lemianti UML įrankių taikymą kuriant programinę įrangą, - UML įrankio naudojimas padeda užtikrinti, kad modelis ir visa jo realizacija yra sinchronizuoti. Be integruoto mechanizmo, užtikrinančio realizacijos ir modelio sinchronizaciją, dizainas ir realizacija yra nesujungti. Realizacijos metu nuolatos paaiškėja geresni sprendimai, taigi dizainas bus keičiamas; jeigu nėra tinkamo grįžtamojo ryšio su modeliu, modelis pasens.

Toliau trumpai apžvelgiamos rinkoje esančių keleto populiarių UML įrankių savybės ir galimybės, verslo procesų reikalavimo modeliavimo ir MDA kontekste. Kadangi visos analizuojamų įrankių savybės ir galimybės nėra svarbios ir mus domina tik kai kurios įrankių galimybės, sudaroma kriterijų lentelė, pagal kurią bus vertinami visi analizuojami įrankiai:

2. Lentelė. UML įrankių vertinimo kriterijai

| **Kriterijus** | **Aprašymas** |
|---|---|
| UML 1.4 palaikymas | Galimybė naudoti UML 1.4 diagramas |
| UML 2.0 palaikymas | Galimybė naudoti UML 2.0 diagramas |
| Robustness diagramos | Galimybė naudoti Robustness diagramas praplečiant UML |
| BPMN palaikymas | Galimybė naudoti Business Process Modeling Notation (BPMN) – standartinius Verslo procesų modelius |
| Kodo inžinerija | Programos kodo generavimas iš diagramų |

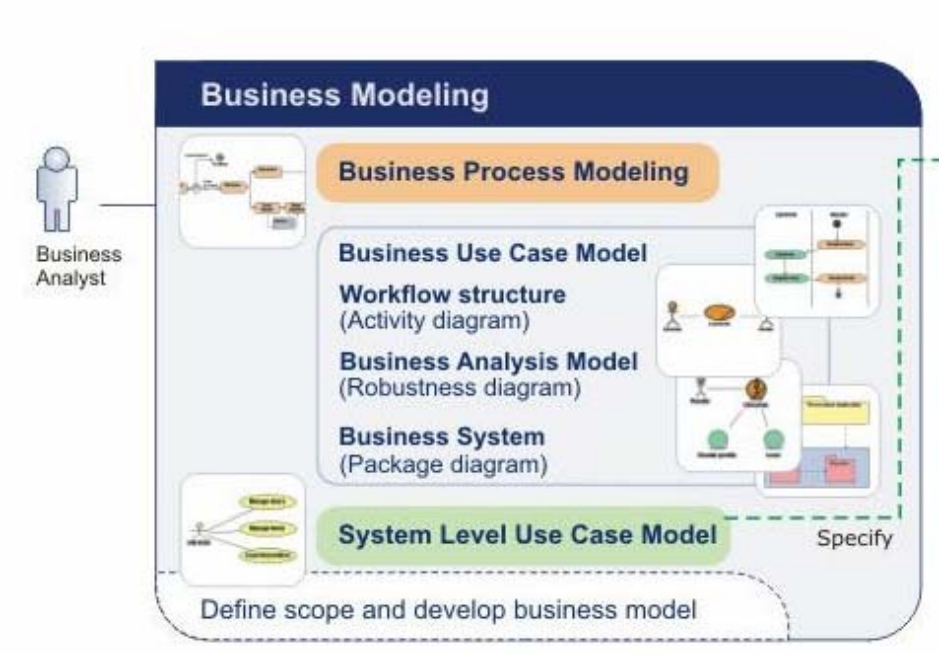
| | Tiesioginė kodo inžinerija | Atvirkštinė kodo inžinerija |
|---|---|---|
| Java palaikymas | Galimybė generuoti kodą iš diagramų | Galimybė generuoti diagramas iš kodo |
| C++ palaikymas | | |
| NET. palaikymas | | |
| CORBA palaikymas | | |
| Visual Basic palaikymas | | |
| MDA palaikymas | Galimybė generuoti PIM ir PSM modelius | |
| Verslo procesų įrankių palaikymas | Integracija su kitais verslo modelių įrankiais | |

## 2.1 Rinkoje egzistuojantys įrankiai

### *2.1.1 MagicDraw UML*

„MagicDraw UML“ programinė įranga - tai UML ir programinės įrangos modeliavimo priemonė pritaikyta komandiniam darbui. Naudodamiesi šia programa, verslo ir kompiuterinių sistemų analitikai, programuotojai, kokybės užtikrinimo specialistai ir dokumentacijos rengėjai gali kurti ir analizuoti objektinio orientavimo sistemas ir duomenų bazes. „MagicDraw UML“ suteikia galimybę iš programos modelio generuoti Java, C#, C++ ir CORBA IDL programinį kodą arba iš jau esamo kodo sudaryti UML diagramas.

Gamintojas deklaruoja padengiantis visus IS vystymo procesus – nuo prieš reikalavimų modelio sudarymą realizuojamą Verslo procesų modeliavimą iki sistemos kodo realizacijos.




**10. pav. MagicDraw galimybės modeliuojant verslo procesus**

Pagal atliktą savybių analizę sudaryta lentelė:

3. Lentelė. MagicDraw galimybių analizė

| **Kriterijus** | **Aprašymas** |
|---|---|
| UML 1.4 palaikymas | + |

| | | |
|---|---|---|
| UML 2.0 palaikymas | + | |
| Robustness diagramos | + | |
| BPMN palaikymas | + | |
| Kodo inžinerija | + | |
| | Tiesioginė kodo inžinerija | Atvirkštinė kodo inžinerija |
| Java palaikymas | + | + |
| C++ palaikymas | + | + |
| NET. palaikymas | + | + |
| CORBA palaikymas | + | + |
| Visual Basic palaikymas | - | - |
| MDA palaikymas | + | |
| Verslo procesų įrankių palaikymas | + | |

Atlikus analizę, galima teigti, kad šis įrankis atlieka daug funkcijų, visada tenkina naujausius rinkos reikalavimus ir verslo poreikius. MagicDraw kodo generavimo inžinerija labiausiai orientuota į klasių diagramų generavimą ir duomenų bazių schemų sudarymą. Tačiau yra ir galimybė generuoti ir Veiklos diagramas iš kodo. Taip pat galima naudoti ir Robustness diagramas.

### *2.1.2 Enterprise Architect*

Rinkoje gana plačiai naudojamas įrankis, realus MagicDraw konkurentas. Galinga sistema, padengianti visą IS kūrimo procesą – nuo reikalavimų iki realizacijos. Turi labai patogią ir paprastą vartotojo sąsają, labai plačias dokumentavimo, įvairių ataskaitų kūrimo galimybes. Labai platus kodo inžinerijos generuojamų programinių platformų pasirinkimas – nuo įprastųjų C++ ir Java, iki PHP ir ActionScript. Konkrečiau įrankio galimybės aptariamos lentelėje:

4. Lentelė. Enterprise Architect galimybių analizė

| **Kriterijus** | **Aprašymas** | |
|---|---|---|
| UML 1.4 palaikymas | + | |
| UML 2.0 palaikymas | + | |
| Robustness diagramos | - | |
| BPMN palaikymas | + | |
| Kodo inžinerija | + | |
| | Tiesioginė kodo inžinerija | Atvirkštinė kodo inžinerija |
| Java palaikymas | + | + |
| C++ palaikymas | + | + |
| NET. palaikymas | + | + |
| CORBA palaikymas | + | + |
| Visual Basic palaikymas | + | + |
| MDA palaikymas | + | |
| Verslo procesų įrankių palaikymas | - | |

Atlikus analizę, galima teigti, kad šis įrankis taip pat yra labai gerai pritaikytas ir orientuotas ir kodo inžineriją ir MDA. Tačiau įrankis nepalaiko Robustness diagramų ir nėra integruotas su kokiu nors verslo procesų modeliavimo įrankiu.

### *2.1.3 Fujaba*

UML modeliavimo įrankis „Fujaba“ leidžia generuoti visapusiškai programos veikimą aprašantį kodą iš UML veiklos diagramų. Tačiau šiame įrankyje tai pasiekiama, surašius programos kodą atitinkamuose UML veiksmuose. Generavimo metu UML veiksme įrašytas programos kodas tiesiog perkeliamas į UML veiksmą atitinkančią vietą, o veiksmai sujungiami tokiu eiliškumu, kokiu jie nurodyti UML veikloje. Faktiškai tai nelabai kuo skiriasi nuo programos kodo užrašymo rankiniu būdu, tik jis įrašomas UML modelyje. Konkrečiau galimybės aptariamos lentelėje:

5. Lentelė. Fujaba galimybių analizė

| **Kriterijus** | **Aprašymas** | |
|---|---|---|
| UML 1.4 palaikymas | + | |
| UML 2.0 palaikymas | - | |
| Robustness diagramos | - | |
| BPMN palaikymas | - | |
| Kodo inžinerija | + | |
| | Tiesioginė kodo inžinerija | Atvirkštinė kodo inžinerija |
| Java palaikymas | + | + |
| C++ palaikymas | - | - |
| NET. palaikymas | - | - |
| CORBA palaikymas | - | - |
| Visual Basic palaikymas | - | - |
| MDA palaikymas | + | |
| Verslo procesų įrankių palaikymas | - | |

Šis įrankis savo galimybėmis smarkiai atsilieka nuo anksčiau aprašytųjų, tačiau jis turi labai aiškią veiklos modelio realizaciją. Tai ko gero vienintelis teigiamas dalykas prieš OO UML įrankius.

### *2.1.4 ArgoUML*

„ArgoUML“ kodo generavimo ir veiklų modeliavimo aspektais nelabai skiriasi nuo „Fujaba“, tačiau yra kiek patogesnis ir paprasčiau naudojamas. Kadangi likusi kodo dalis įrašoma rankiniu būdu, tai iš papildomos objektų sukūrimo bei sunaikinimo operacijų abstrakcijos naudos beveik nėra.

Šis įrankis yra nemokama, atviro kodo programa, vadinasi kiekvienas išmanantis UML specifikaciją ir žinantis metodologiją specialistas, gali pritaikyti įrankį savo reikmėms.

6. Lentelė. ArgoUML galimybių analizė

| **Kriterijus** | **Aprašymas** | |
|---|---|---|
| UML 1.4 palaikymas | + | |
| UML 2.0 palaikymas | - | |
| Robustness diagramos | - | |
| BPMN palaikymas | - | |
| Kodo inžinerija | + | |
| | Tiesioginė kodo inžinerija | Atvirkštinė kodo inžinerija |

| | | |
|---|---|---|
| Java palaikymas | + | + |
| C++ palaikymas | - | - |
| NET. palaikymas | - | - |
| CORBA palaikymas | + | + |
| Visual Basic palaikymas | - | - |
| MDA palaikymas | + | |
| Verslo procesų įrankių palaikymas | - | |

Savo savybėmis ir pajėgumu šis įrankis panašus į Fujaba. Taigi ir šis įrankis gerokai atsilieka nuo komerciškai sėkmingų ir nuolat vystomų didžiųjų šios srities produktų. Tačiau didelis ArgoUML pranašumas yra tas, kad tai – atviro kodo programa, kurią galima modifikuoti, pritaikant savo reikmėms.

### *2.1.5 Išvados*

Aptarėme žinomus ir galingus įrankius ir šalia palyginimui buvo parinkti mažesni sprendimai, orientuoti į vieną modeliavimo sritį ar metodiką.

Atlikus pasirinktų įrankių analizę, aiškiai matyti, kad įrankiai kol kas nėra pakankamai orientuoti į IS vartotojo reikalavimų modelį verslo procesų kontekste. Nėra stiprių loginių ryšių tarp Verslo procesų modelio, atvaizduojamo į Veiklos modelį, ir UCM modelio. Nėra jokių modelių transformavimo galimybių iš vienos diagramos į kitą. Taigi šią nišą būtų galima mėginti užpildyti eksperimentiniu sprendimu.

## 2.2 Galimi realizacijos būdai

- **ArgoUML naudojant Java**

Kadangi šis įrankis yra atviro kodo programa, prašyta Java kalba, yra visos galimybės redaguoti kodą ir sukurti planuojamą sprendimą. Sistema yra maža, aiškiai suprantama, vadinasi ir perprasti kodą bei realizuoti suprogramuotą įskiepį gali būti nesudėtinga.

- **MS Visio naudojant Visual Basic**

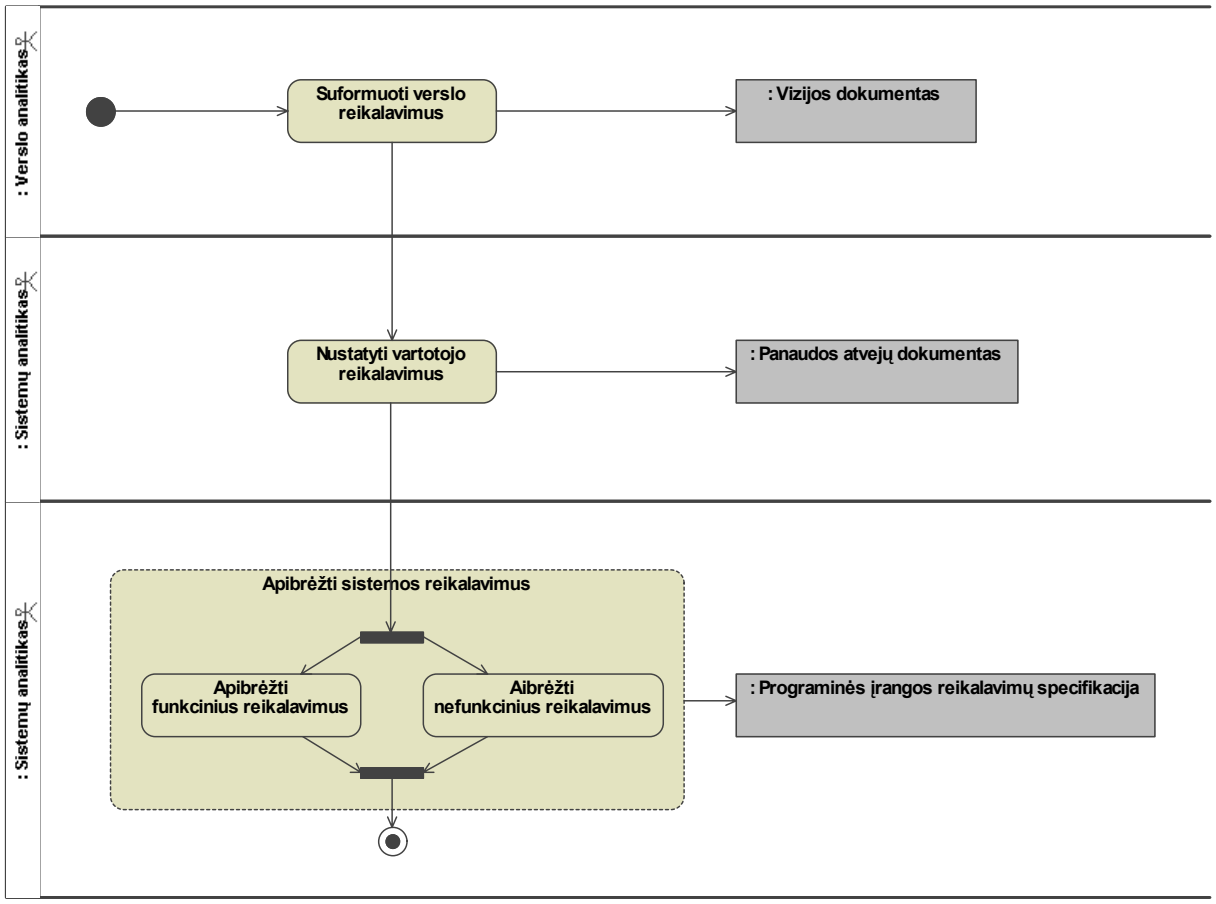
Ne mažiau realus variantas yra ir šis, populiarusis Microsoft gaminys, kuris neturi nė pusės anksčiau paminėtųjų įrankių savybių, tačiau yra paprastas, lengvai suderinamas su daugeliu programų. MS Visio įrankio aplinkoje yra galimybė koduoti savo adaptacijas Visual Basic programavimo kalba.

- **Magic Draw UML naudojant Java įskiepį (plug-in)**

Ko gero populiariausias ir vienas geriausių rinkoje UML įrankių, nuolat tobulinamas ir suderinimas su begale metodologijų, kalbų ir platformų. Šis įrankis pasirinktas praktinei realizacijai įgyvendinti, pagrindžiant Veiklos ir vartotojo reikalavimų modelių sąsają.

# 3. Vartotojo reikalavimų modelio sudarymo metodas

Reikalavimų valdymo procesuose IS kūrime, galima apibrėžti tokią seką, kurios atitinkami etapai gali būti modeliuojami tam skirtomis diagramomis:




**11. pav. Skirtingų lygių reikalavimų analizės etapai**

Verslo reikalavimų suformavimo procese naudosime Veiklos modelį, kad tinkamai ir vaizdžiai atvaizduoti verslo procesus. Vartotojų reikalavimų nustatymo etape naudosime panaudojimo atvejų UCM modelį. O apibrėžiant reikalavimus IS, Veiklos modelio pagalba pritaikysime UCM modelį ir patikrinsime jo išbaigtumą naudojant Robustness diagramą.

## 3.1 Sąsajomis grįstas IS projektavimas

Į verslo procesus ir vartotojo reikalavimus orientuotame IS kūrime sąsajos paprastai atlieka pagrindinį vaidmenį. Nors objektinis projektavimas yra paprastesnis negu sąsajų naudojimas, sąsajos leidžia naudoti pakartotinį panaudojimą be tvirtų sujungimų, o tai jau

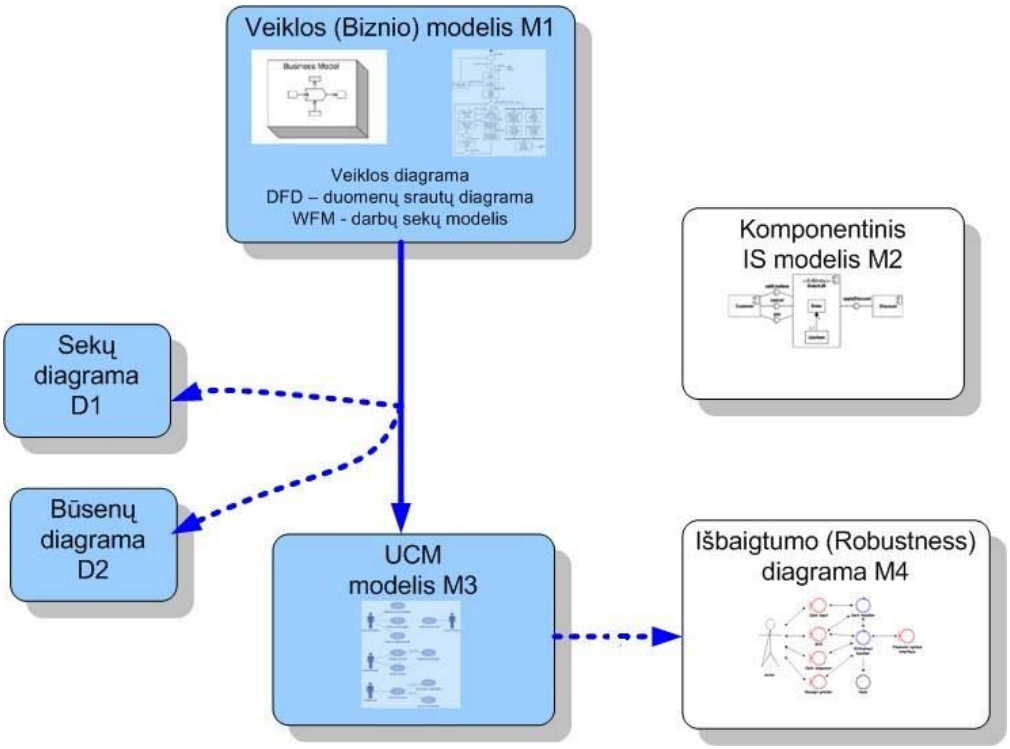
didžiulis pranašumas prieš objektinį projektavimą, nes jis remiasi tvirtu klasių apjungimu, paveldėjimu, kas daro kodą pažeidžiamą ir sunkiau panaudojamą. Žinoma, tai nereiškia, kad paveldėjimas tarp klasių objektiniame projektavime nėra naudingas. Tačiau mūsų laikais, nuolat didėjant reikalavimas IS ir kylant apimtims, atrastas kitas būdas efektyviau realizuoti pakartotinį panaudojimą, neprarandant informacijos plėtimo galimybių, Taip ir buvo sukurtas sąsajomis grįstas projektavimas ir programavimas [8].

Naudojant sąsajas IS kūrimo ciklas tampa ilgesnis,, tačiau jis gali būti lengviau pritaikomas nuolat besikeičiant verslo procesams ir poreikiams. Sąsajos leidžia lengviau palaikyti ir plėsti informacijos sistemas, taigi, naudinga vystyti metodą, kuris padėtų teisingai projektuoti IS naudojant sąsajas [8].

### 3.2 Galimos naudojamų modelių sąsajos

Aptarus IS projektavimą sąsajų pagalba, tikslinga pavaizduoti schemą, nurodant ryšius tarp modelių. Planuojama realizuoti sąsają tarp Veiklos ir UCM modelių, panaudojant ir Komponentinį IS modelį bei Išbaigtumo (Robustness) diagramas. Komponentinis modelis gali būti generuojamas iš Veiklos (biznio) modelio – darbų sekų arba veiklos (activity) diagramos ir vaizduojamas kaip papildomas biznio modelis, kuris vėliau gali būti susistemintas kaip IS architektūros ir komponentų modelis, kurio pagalba Veiklos modelis gali būti atvaizduojamas į UCM. UCM modelio išbaigtumas ir korektiškumas gali būti tikrinamas Robustness diagramų pagalba.

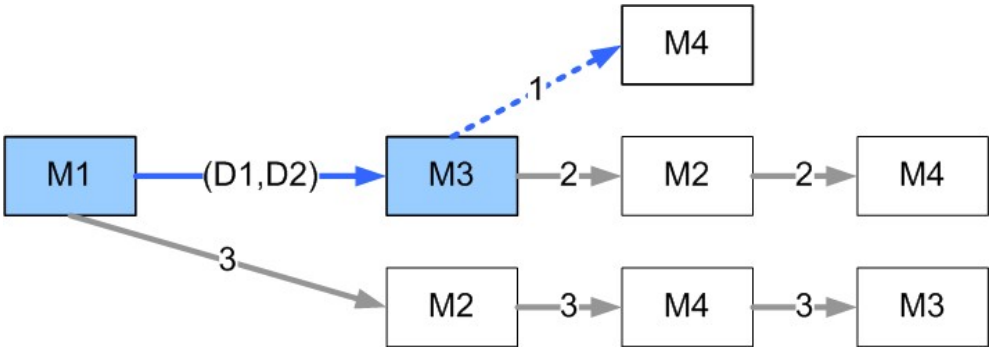
Pateikiu bendrą veiklos procesų specifikavimo modelių diagramą ir galimas jų sąsajas:



**12. pav. Nagrinėjamų modelių galimos sąsajos**

Rodyklėmis parodytos sąsajos žymi galimus pateiktų modelių sąsajų variantus. Matome, jog identifikuojant verslo procesus, be Veiklos ir UCM modelių, galima naudoti ir Komponentinį modelį, kuris gerokai papildo UCM aprašytą biznio logiką ir apjungia Veiklos modelį su UCM. O UCM modelio išbaigtumui ir teisingumui tikrinti galima naudoti Išbaigtumo (Robustness) diagramą. Taip pat akivaizdžiai matosi, kad norint išlaikyti stiprų ryšį tarp Veiklos modelio ir UCM, labai svarbu panaudoti sekų arba būsenų diagramas, tam kad išsaugoti veiksmų seką, nes joks UCM modelis sekos neišsaugo. Tam praktinėje realizacijoje bus naudojamos sekų diagramos, o komponentinis modelis ir išbaigtumo diagramos toliau nenagrinėjamos.

Tam, kad aiškiau pateikti būsimus sąsajų variantus, pateikiama diagrama:




**13. pav. Galimos modelių sąsajų generavimo sekos**

Pagal šia diagramą matosi, kad tolimesniame darbe bus remiamasi viena ir aukščiau parodytų sąsajų – bus atliekamas Veiklos modelio (UML Activity diagrama) sudarymas kaip verslo procesų reikalavimo modelio ir jo generavimas į panaudojimo atvejų modelį (UCM), kontekste naudojant ir sekų arba būsenų diagramą tam, kad išsaugoti veiklos modelio vaizduojamų veiksmų seką. UCM pilnumas ir korektiškumas gali būti tikrinamas išbaigtumo (Robustness) diagramų pagalba. Šis kelias paveikslėlyje nurodytas kaip M1→M3→M4.

## 3.3 Vartotojo reikalavimų modelio transformavimas iš veiklos modelio

### *3.3.1 Formalizuotas veiklos meta – modelio aprašymas abstrakčiosios algebros pagrindu [10]*

Toliau pateikiamas teorinis formalizuotas VM ir UCM metamodelių aprašymas, kuris reikalingas UCM generavimo algoritmui aprašyti. VM metamodelį aprašysime abstrakčiosios algebros pagrindu kaip algebrinę sistemą M (Malcev, 1970):

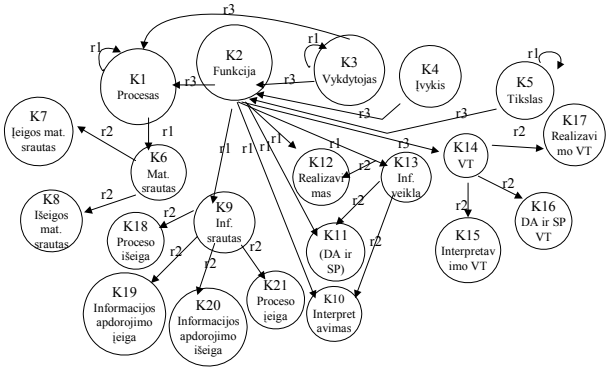
**M=<K, R>;** kur **M**– veiklos metamodelis; **K**– klasių aibė, kur **K**={K1, K2,…, K21}, **R**– ryšių aibė, kur R={r1, r2, r3}.

Kiekvieno klasių aibės **K** elemento Kn sudėtis apibrėžiama taip: Kn=<{$an_1$, $an_2$,…,$an_k$}, {$mn_1$, $mn_2$,…,$mn_l$}>, kur { $an_1$, $an_2$,…,$an_k$ }- elemento Kn atributai, { $mn_1$, $mn_2$,…,$mn_l$ }- elemento Kn metodai. [10]

Veiklos metamodelio **M1** sudėtis yra tokia:

**M1**=<{K1, K2,...,K21}, {r1, r2, r3}>, kur K1– klasė *Procesas*, K2– klasė *Funkcija*, K3– klasė *Vykdytojas*, K4– klasė *Įvykis*, K5– klasė *Tikslas*, K6– klasė *Materialus srautas*, K7– klasė *Įeigos materialus srautas*, K8– klasė *Išeigos materialus srautas*, K9– klasė *Informacinis srautas*, K10– klasė *Interpretavimas*, K11– klasė *Duomenų apdorojimas ir sprendimų priėmimas* (DA ir SP), K12– klasė *Realizavimas*, K13– klasė *Informacinė veikla*, K14– klasė *Veiklos taisyklės* (VT), K15– klasė *Interpretavimo veiklos taisyklės*, K16– klasė *Duomenų apdorojimo ir sprendimų priėmimo veiklos taisyklės* (DA ir SP VT), K17– klasė *Realizavimo veiklos taisyklės*, K18– klasė *Proceso išeiga*, K19– klasė *Informacijos apdorojimo įeiga*, K20– klasė *Informacijos apdorojimo išeiga*, K21– klasė *Proceso įeiga*, r1– agregavimo santykis, r2– apibendrinimo santykis, r3– asociacija. Veiklos metamodelio grafinė schema pateikiama 2 paveiksle [10]:

*Šaltinis: Lopata A., Gudas S. Veiklos modeliu grindžiamas vartotojo poreikių specifikavimas. KTU Informacijos sistemų katedra, VU KHF Informatikos katedra, 2005.*



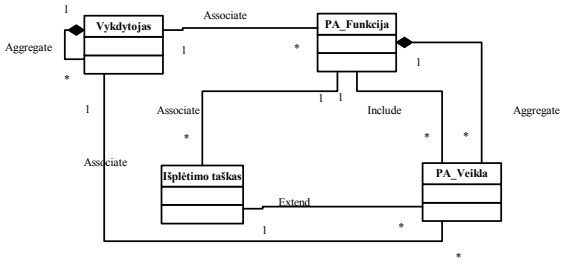

**14. pav. Veiklos metamodelio M1 schema**

### *3.3.2 UCM generavimas pasirinktai veiklos modelio funkcijai [10]*

Detaliau panagrinėsime UCM, sudaromo konkrečiai veiklos funkcijai, generavimo pagrindinius žingsnius [10].

Vienai veiklos modelyje specifikuotai funkcijai generuojamą UCM vadinsime “funkcijos UCM”. Funkcijos UCM metamodelį sudaro klasės *Vykdytojas*, panaudojimo atvejis *PA_Funkcija*, panaudojimo atvejis *PA_Veikla* ir *Išplėtimo taškas,* 3 tipų ryšiai *Associate*, *Include* ir *Generalize*. Funkcijos UCM metamodelis pateiktas paveiksle:

*Šaltinis: Lopata A., Gudas S. Veiklos modeliu grindžiamas vartotojo poreikių specifikavimas. KTU Informacijos sistemų katedra, VU KHF Informatikos katedra, 2005.*



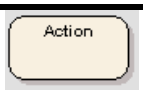
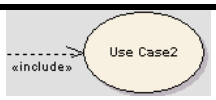
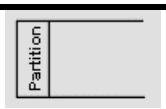
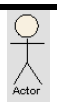

**15. pav. Funkcijos UCM metamodelis**

Remiantis šiais aptartais elementais (funkcijomis) ir ryšiais tarp jų, bus sudaromas algoritmas, kuriuo remiantis atliekama UCM modelio transformacija iš Veiklos modelio.

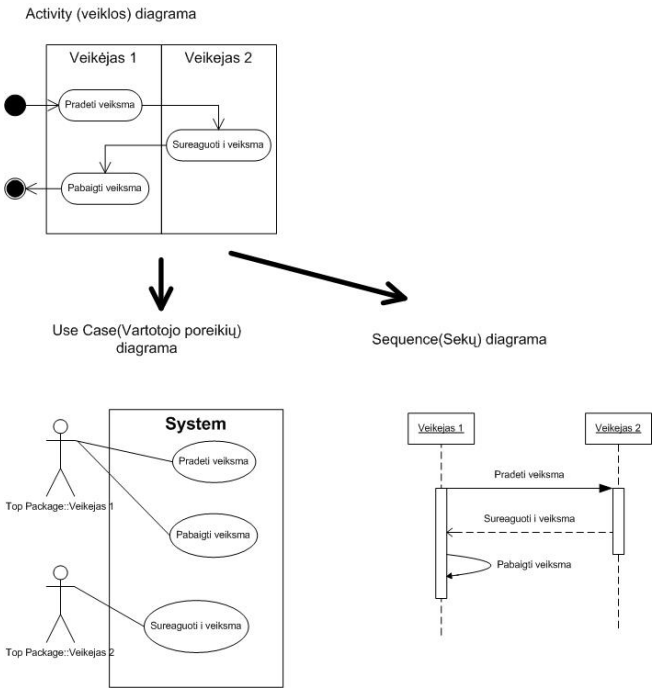
### *3.3.3 Veiklos modelio ir UCM elementų loginis ryšys*

Aprašysime veiklos modelio ir UCM loginį ryšį ir transformavimo algoritmą, kaip vieno modelio elementai transformuojami į kito modelio elementus. Atskiri veiklos modelio elementai transformuojami į atitinkamus UCM modelio elementus išlaikant bendrą jų ryšį ir logiką. Toliau lentelėje pateikiama kokie elementai į kokius transformuojami:

7. lentelė. Veiklos modelio elementų atvaizdavimas į atitinkamus UCM elementus

| **Veiklos modelio elementas** | **Grafinis vaizdas** | → | **Transformuotas elementas UC modelyje** | **Grafinis vaizdas** |
|---|---|---|---|---|
| Veikla (Activity) |   | | Tėvinis panaudojimo atvejis (Use-Case) |   |
| Veiksmas (Action) |   | | Panaudojimo atvejis (Use-Case) su ryšiu <<include>> iš tėvinio panaudojimo atvejo |   |
| Veikėjas/padala (Partition) |   | | Veikėjas (Actor) |   |

Duomenų transformacijos pavyzdžiai pateikiami žemiau:




**16. pav. Funkcijos UCM metamodelis**

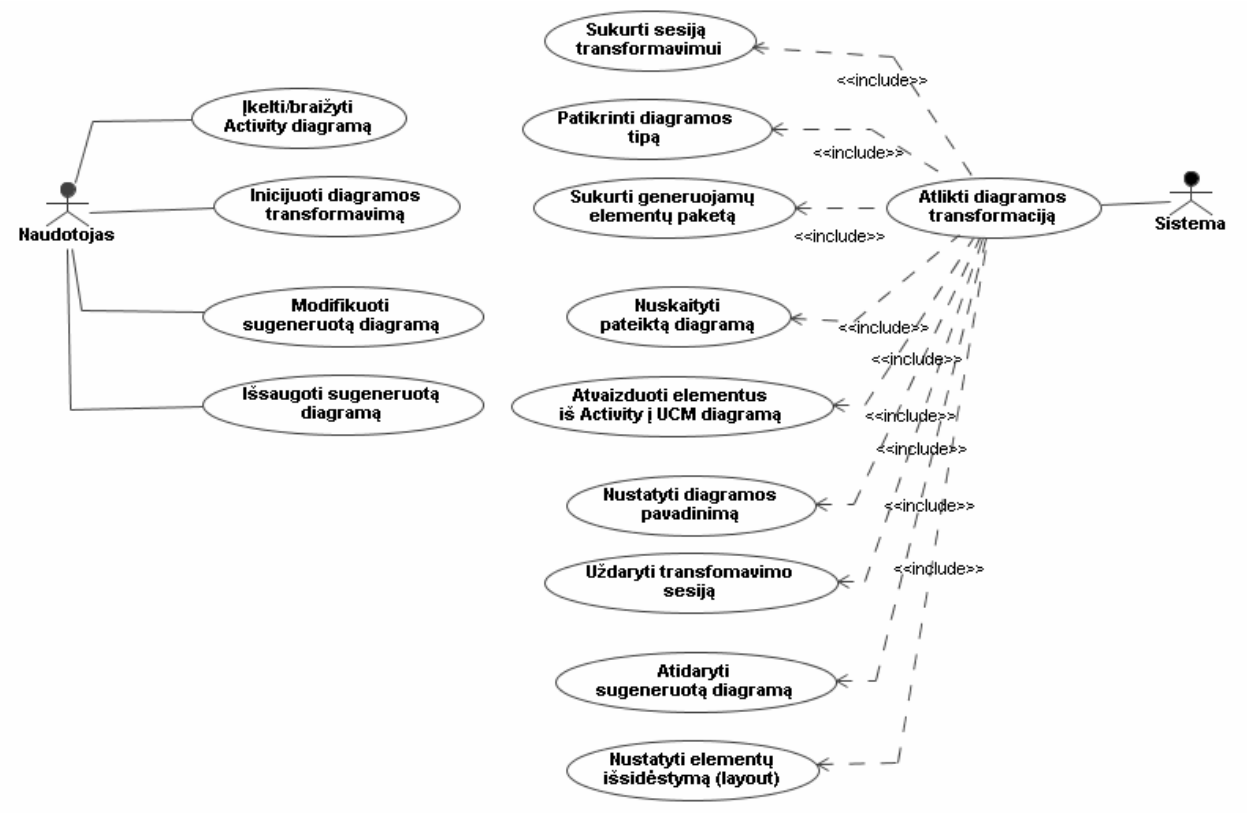
# 4. Modelių transformacijos prototipo realizacija

## 4.1 Veiklos modelio transformavimo į UCM modelį realizacijos projektas

### *4.1.1 Sprendimo funkcionalumo nustatymas*

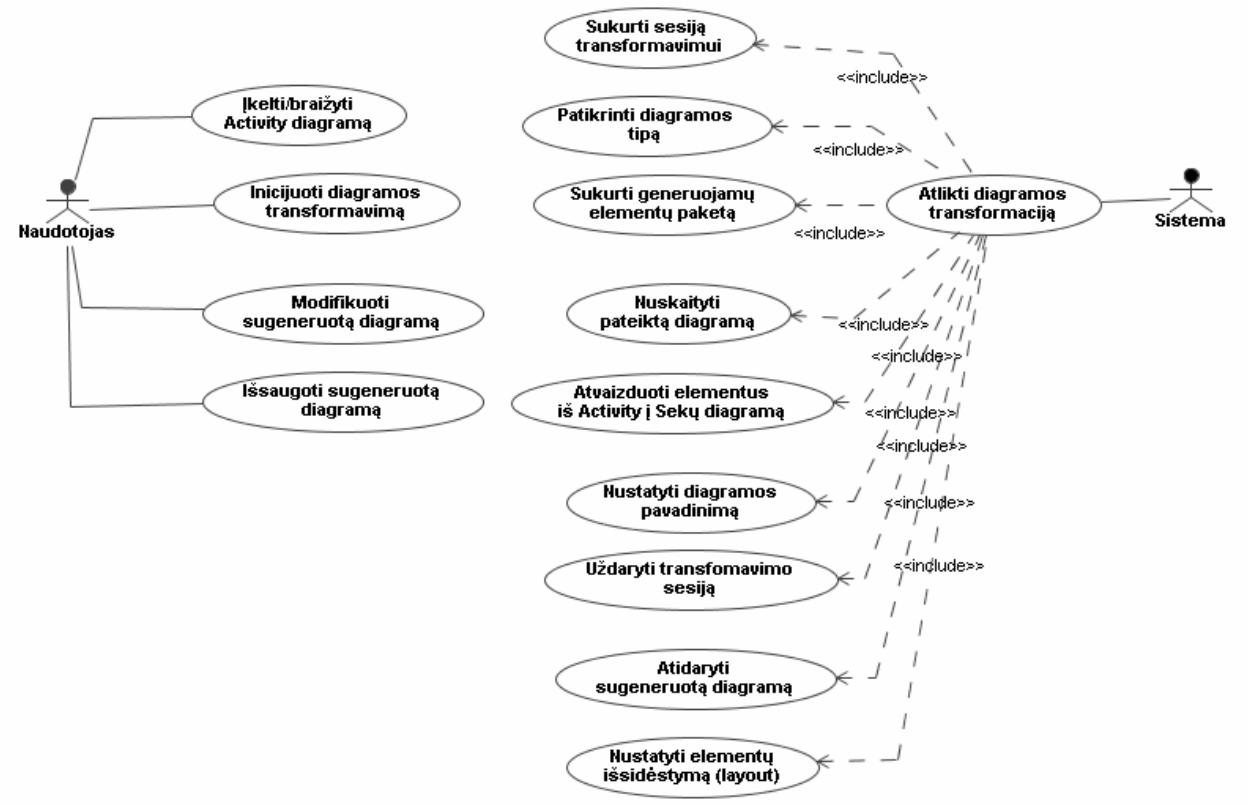
Veiklos modelio transformavimas į UCM modelį yra automatizuojamas, vartotojas tik įkels reikiamą Veiklos modelį į sistemą ar nubraižys jį, toliau inicijuodamas transformacijos procesą, kurio rezultate sugeneruojamas UCM modelis. Visą modelio transformacija atliekama sistemos viduje ir naudotojas mato tik galutinį rezultatą – sugeneruotą UCM modelį, kurį, esant poreikiui galima koreguoti išsaugoti. Taip pat vykdoma ir transformacija į sekų arba būsenų diagramas, nes UCM diagrama neišsaugo veiksmų sekos, vaizduojamos Activity diagramoje. Tam ir naudojamos minėtosios diagramos.

Toliau pateikiamos sprendimo realizacijos funkcijų pasiskirstymo diagramos (aprašai 1 Priede. Modelių transformacijos sprendimo panaudojimo atvejų aprašymai).




**17. pav. Veiklos modelio transformacijos į UCM diagramą panaudojimo atvejų diagrama**

Kadangi UCM nesaugo informacijos apie veiksmų seką, generuojama sekų diagrama:




**18. pav. Veiklos modelio transformacijos į Sekų diagramą panaudojimo atvejų diagrama**

Būsenos diagramos transformacijos panaudojimo atvejų diagrama nepateikiama, nes ji analogiška pateiktosioms.

Veiklos diagramą, kuri transformuojama į kitą modelį, pasirenka ar nubraižo naudotojas, o transformavimo veiksmus atlieka sistema, pateikianti galutinį rezultatą taip atmetant bet kokią klaidos galimybę.

Naudotojas kontroliuoja šiuos duomenis:

- pateikiamą Veiklos diagramą;
- sugeneruoto poreikių modelio elementų išdėstymą;

Sistema kontroliuoja šiuos duomenis:

- pateiktos diagramos meta-duomenis;
- generuojamų elementų aibę;
- sugeneruotos diagramos išvedimo rezultatus;
- sugeneruoto poreikių modelio elementų išdėstymą;.

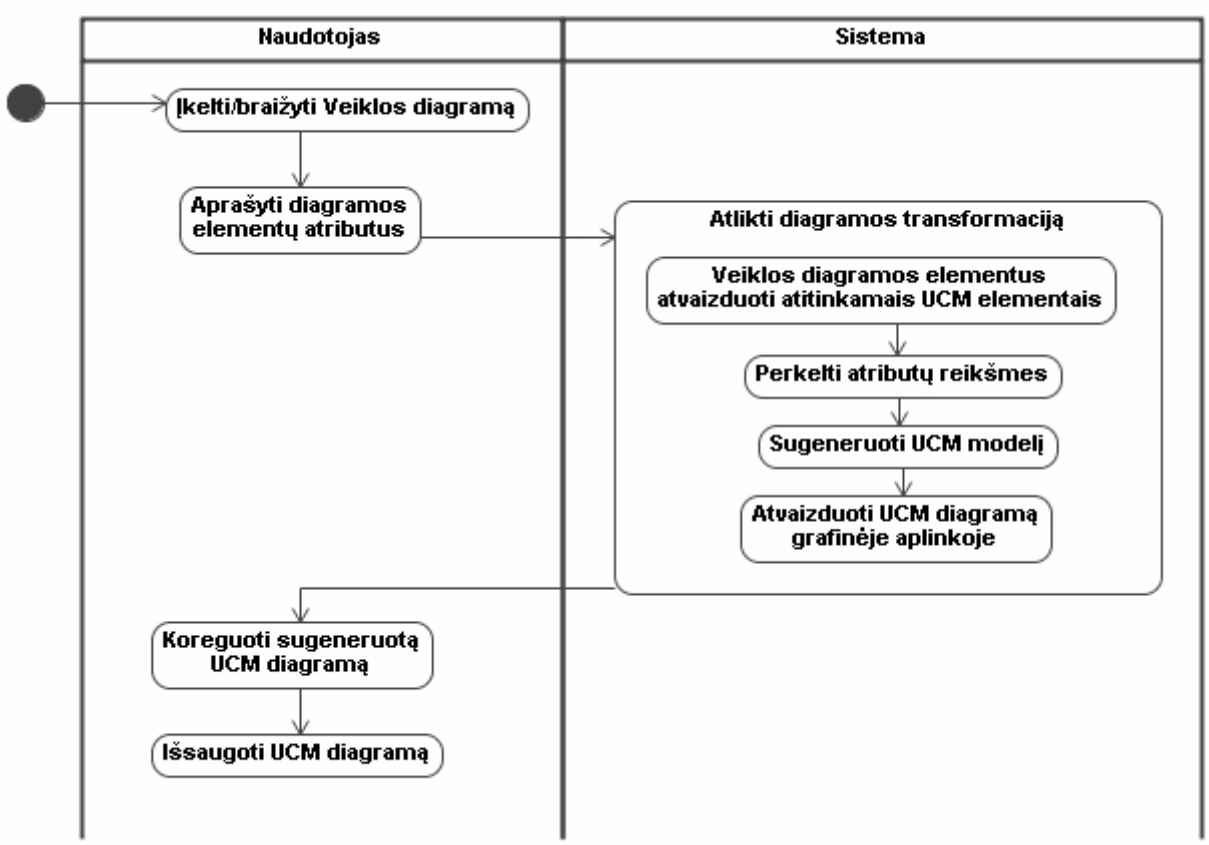
### *4.1.2 Sprendimo veiklos diagrama*

Vartotojo reikalavimų IS modelio transformavimas iš Veiklos modelio paremtas atitinkamų vieno modelio elementų atvaizdavimu kito modelio atitinkamais elementais. Tokiu būdu iš Activity diagramos sugeneruojama Use-case diagrama išvengiant klaidų ir laikantis

UML notacijos standartų. Užtikrinti visiško transformuoto modelio tinkamumo nėra įmanoma, todėl naudotojui paliekama galimybė koreguoti sugeneruotą modelį.

Naudotojas, atliekantis modelių transformacijas, nurodo reikiamą veiklos modelį, o sistema sugeneruoja rezultatą - UCM modelį.

Modelių transformavimo procesas:



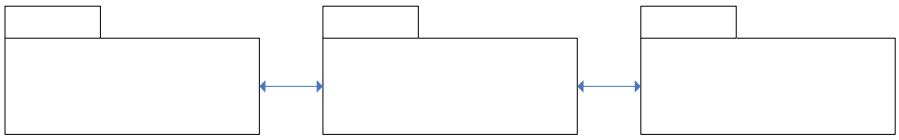

**19. pav. Veiklos modelio transformacijos į UCM veiklos diagrama**

Analogiškai vykdoma ir Veiklos modelio transformacija į sekų ir būsenų diagramas.

Pateikta veiklos diagrama parodo viso generavimo proceso eigą, veiksmų seka yra griežtai apibrėžta.

### *4.1.3 Sistemos loginė architektūra*

Sistemos architektūros kūrimui panaudotas tipinis trijų lygių architektūros modelis. Remiantis šiuo modeliu rekomendacijų formavimo sistemą sudaro vartotojo, veiklos bei duomenų paslaugos, kurios sugrupuojamos į atskirus paketus.

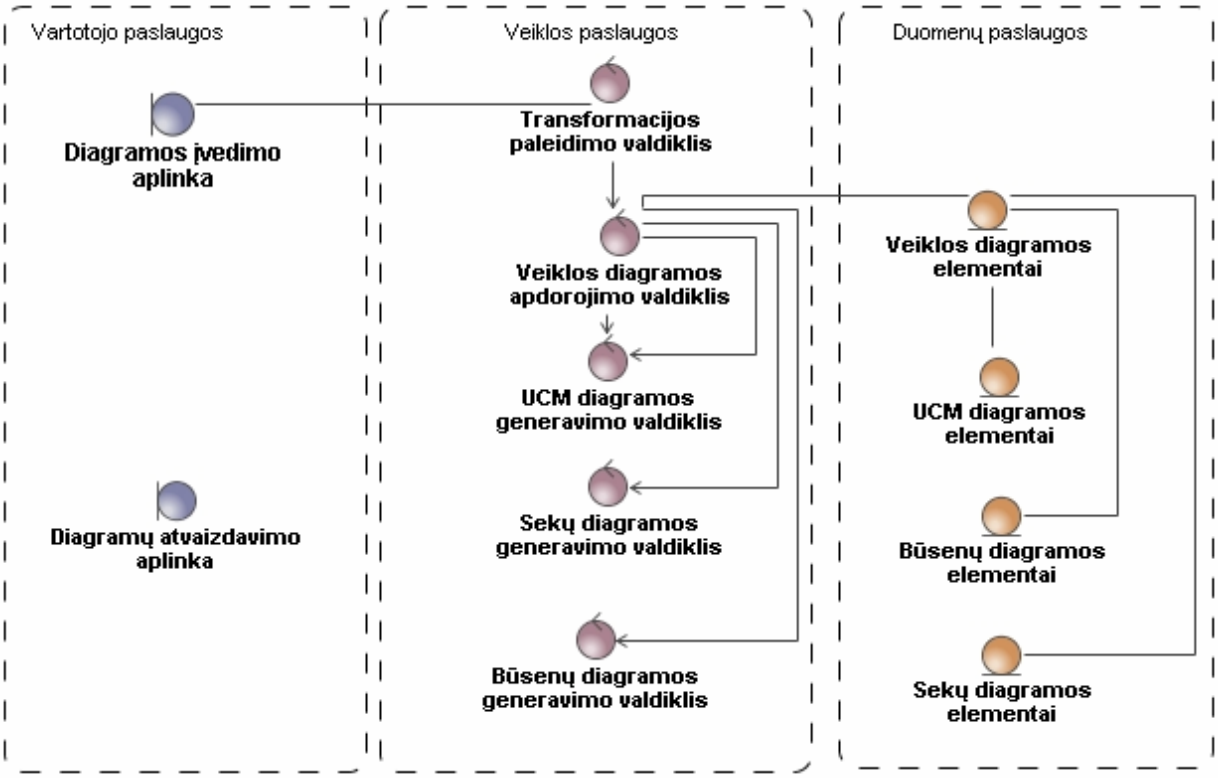


**20. pav. Sistemos loginė architektūra**

Vartotojo paslaugos apima grafinę sąsają, per kurią vartotojas sąveikauja su sistema. Veiklos paslaugos yra programinis įskiepis, kuris atlieka vartotojo užklausą, transformuoja diagramas iš pateiktos Activity diagramos. Duomenų paslaugos – tai sistemos elementai skirti laikinai saugoti duomenims, modelių transformacijos metu.

Vartotojas grafinėje aplinkoje (vartotojo paslaugos) įkelia arba nubraižo savo diagramą, įskiepio valdiklis (veiklos paslaugos) apdoroja informaciją ir vykdo transformaciją, apdorodamas duomenis sistemos viduje (duomenų paslaugos), gauna reikiamus rezultatus, kuriuos per grafinę vartotojo aplinką (vartotojo paslaugos) pateikia vartotojui.

Remiantis panaudojimo atvejų diagrama pav., identifikuojamos vartotojo funkcijos („*Vartotojo paslaugos*“), pagal kurias nustatomos vartotojo sąsajos klasės, jos pateikiamos žemiau. Atsižvelgiant į veiklos diagramą, identifikuojami valdikliai, kurie realizuoja sistemos funkcijas („*Veiklos paslaugas*“). Jie susieti su vartotojo klasėmis.

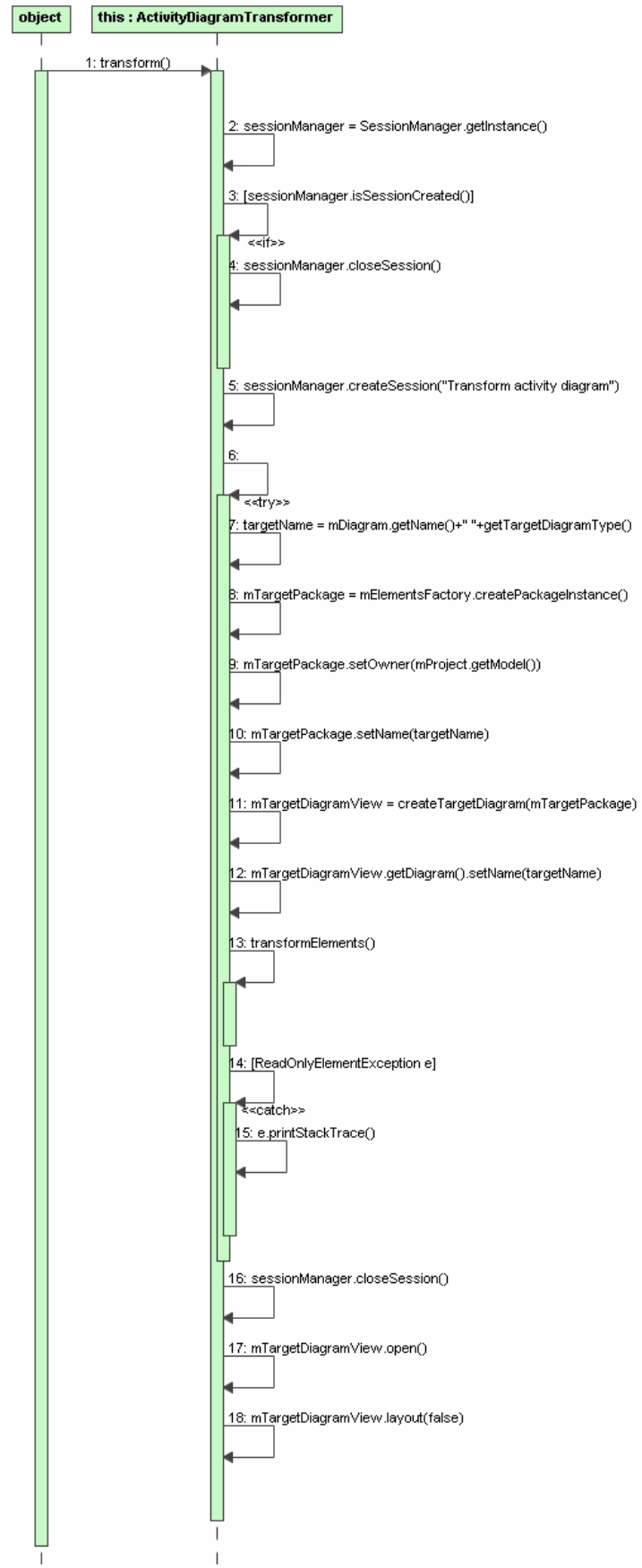



**21. pav. Sistemos klasių diagrama**

### *4.1.4 Sistemos panaudojimo atvejų sekų diagrama*

Panaudojimo atvejis iš Panaudojimo atvejų diagramos detalizuojamas sekų diagramomis. Pagrindinio panaudojimo atvejo („Atlikti diagramos transformaciją“ ) sekų diagrama pateikta žemiau.

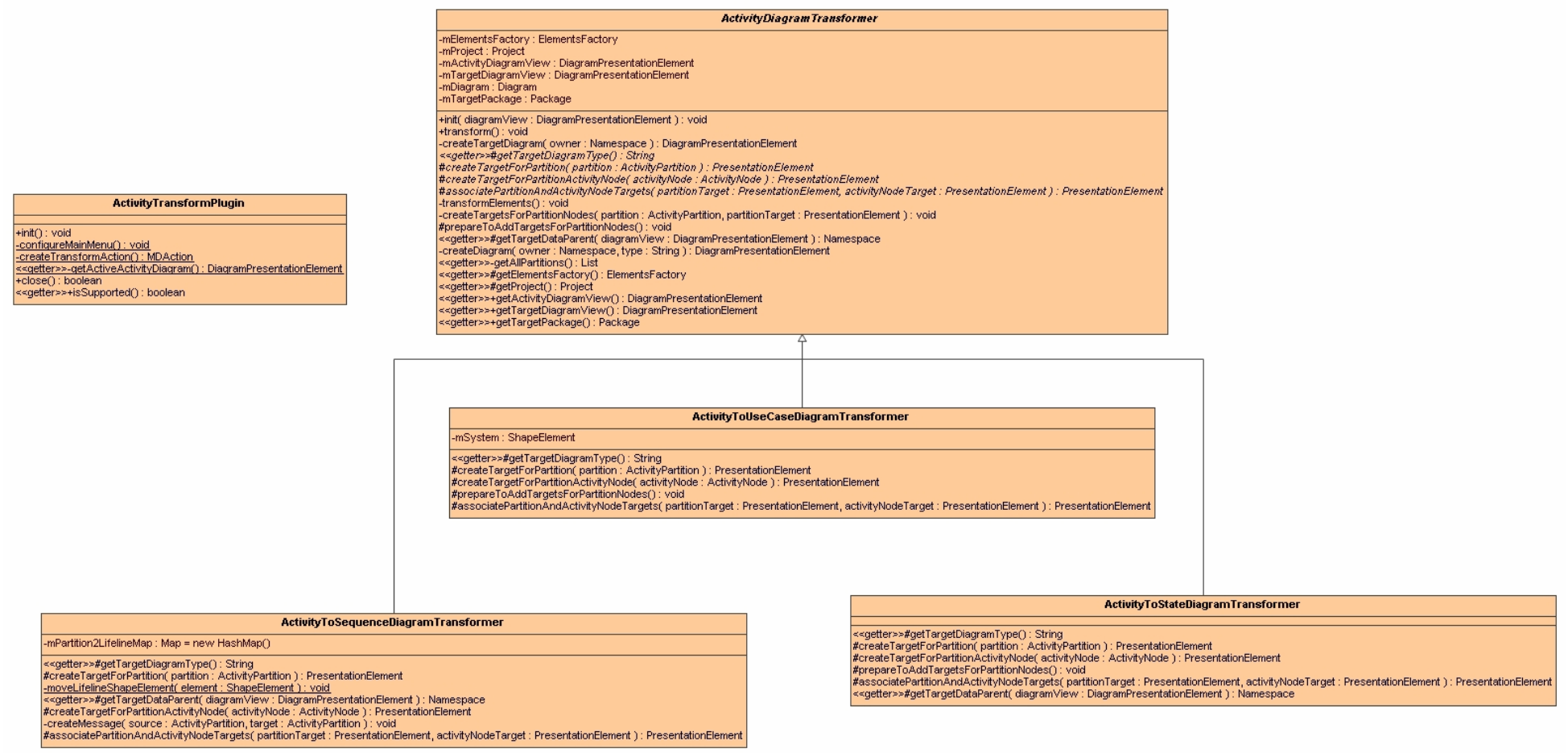
Sekų diagrama sugeneruota MDA besiremiančios kodo inžinerijos principu iš sukurto realizacijos kodo.




**22. pav. Panaudojimo atvejo „Atlikti diagramos transformaciją“ specifikavimas sekų diagrama**

### *4.1.5 Sistemos klasių diagrama*

Diagramų transformavimo klasių diagrama pateikta žemiau:

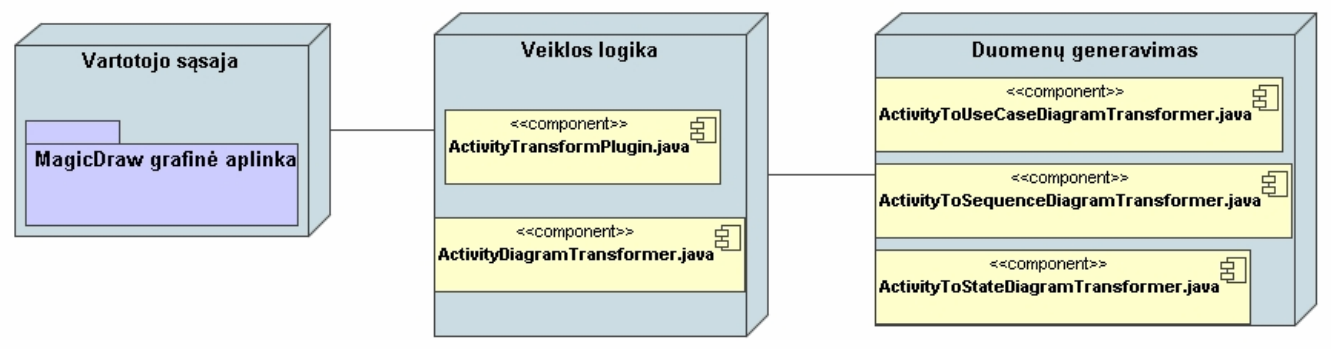



**23. pav. Transformavimo sprendimo klasių diagrama**

## 4.2 Veiklos modelio transformavimo į UCM realizacija

### *4.2.1 Sprendimo komponentų diagrama*

Atsižvelgiant į sistemos funkcionalumą, bei atlikus galimų sprendimų analizę, sudaroma komponentų diagrama, realizuojanti projektuojamo sprendimo architektūrą.

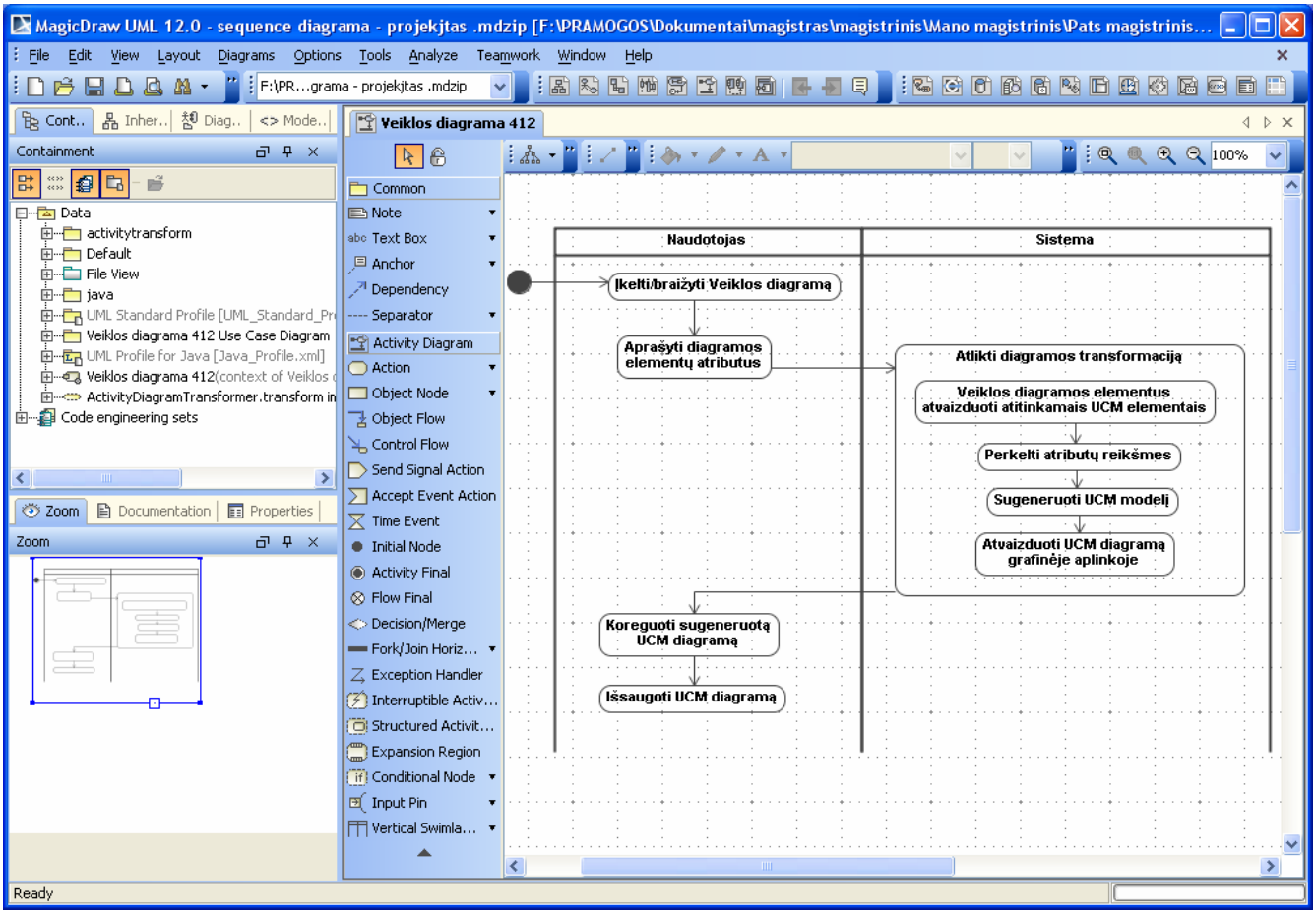



**24. pav. Projektuojamo sprendimo komponentų diagrama**

### *4.2.2 Sprendimo realizacijos prototipo duomenys ir rezultatai*

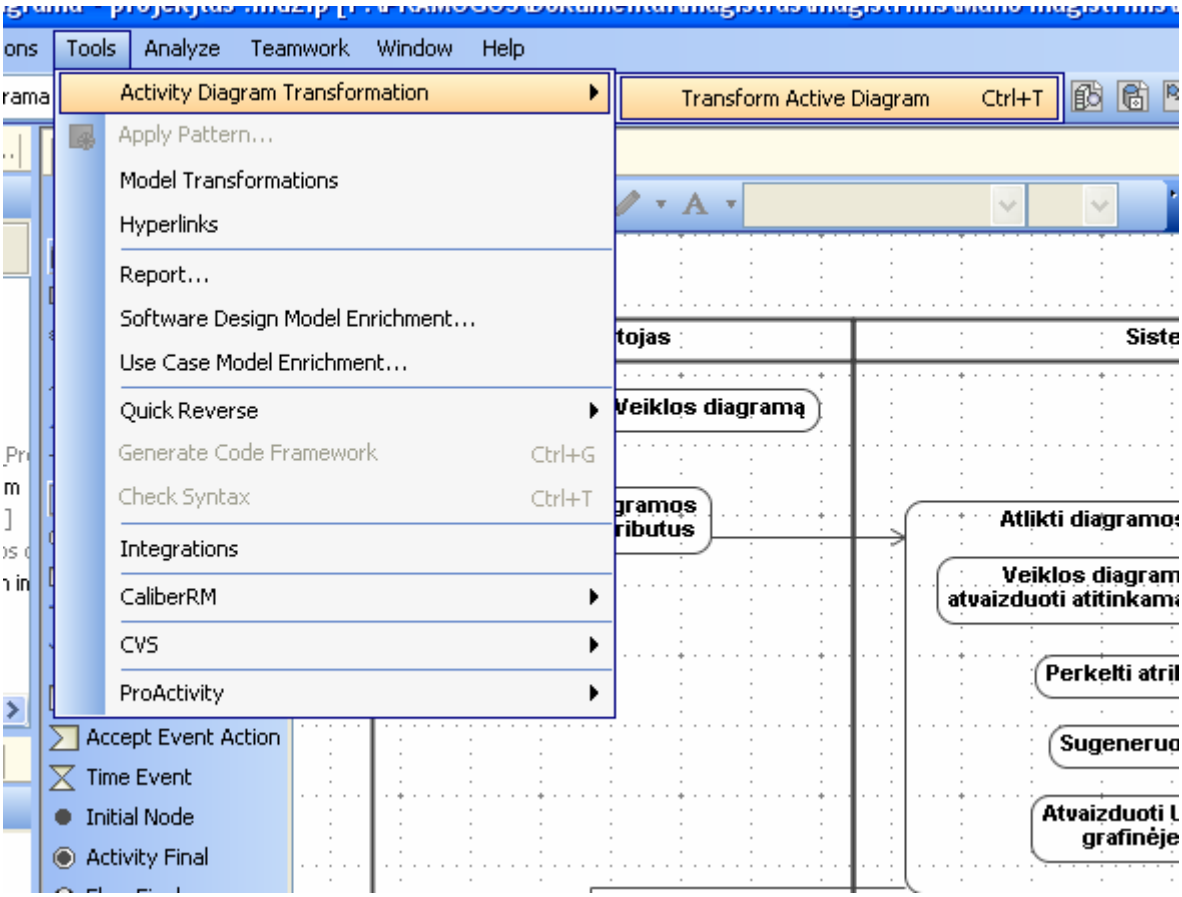
Sprendimo realizacijos prototipas buvo orientuotas į UML įrankio grafinėje aplinkoje kuriamą įskiepį. Galutinei realizacijai pasirinktas MagicDraw įrankis, kurio aplinkoje ir realizuotas diagramų transformavimo sprendimas.

Naudotojas MagicDraw aplinkoje įkelia arba nubraižo Veiklos modelį analizuojamam verslo procesui ar veiklai. Pavyzdyje naudosime šiam darbui sukurtą ir panaudotą veiklos diagramą, aprašančia patį transformavimo modelį. Tarkime, naudotojas nubraižė naują veiklos diagramą. Pradiniai duomenys tokie:



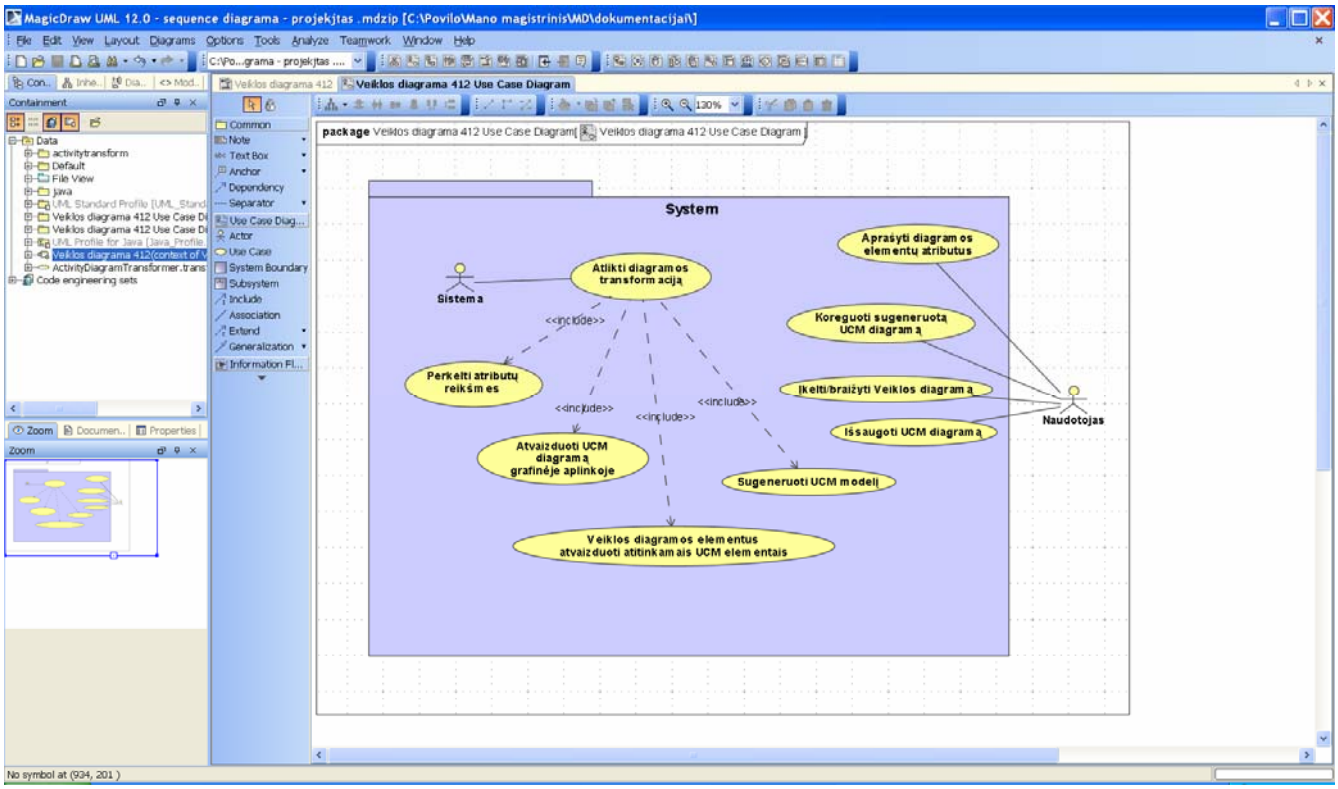

**25. pav. MagicDraw aplinkoje nubraižyta Veiklos diagrama**

Nubraižius ar įkėlus diagramą, naudotojas pasirenka transformavimo veiksmą. Meniu juostoje, pasirenka Tools->Activity Diagram Transformation ->Transform Active Diagram. Paprastesniu būdu, naudotojas tiesiog gali spausti Crtl+T komandą ir transformavimas bus inicijuotas.




**26. pav. Modelio transformavimo komanda**

Po modelio transformacijos, sistema iš Activity diagramos sugeneruoja ir atvaizduoja UCM diagramą. Rezultatas pateikiamas žemiau:




**27. pav. Sugeneruotos diagramos vaizdas**

Naudotojas gali pakoreguoti ar papildyti sugeneruotą diagramą ir ją išsaugoti naujame ar atidarytame projekte.

Analogiškai generuojamos sekų ir būsenų diagramos, kurios papildo UCM, nes šis modelis nesaugo informacijos iš Veiklos modelio apie veiksmų atlikimo seką.

# Išvados

1. Pagal atliktą esamų modeliavimo įrankių analizę, nustatyta, kad pasiūlytas sprendimas praplečia įrankio galimybes ir jo panaudojimą automatizuojant modelių transformacijas.
2. Buvo atlikta rinkos ir joje esančių UML modeliavimo įrankių analizė. Išnagrinėtos analizuotų įrankių galimybės ir trūkumai,
3. Atlikus analizę, išsiaiškinta, kad nei vienas įrankis neatlieka veiklos ir vartotojo reikalavimų modelių transformacijos.
4. Atsižvelgiant į gautas analizės išvadas, pasiūlytas sprendimas, transformuojantis Veiklos modelį į vartotojo reikalavimų IS modelį (UCM), remiantis UML standartu, tam panaudojant UML įrankį.
5. Darbe pasiūlytas pagal pradinę vartotojo sąlygą atliekamas automatizuotas vartotojo reikalavimų modelio (UCM) generavimas, kuris atitinka naujausiomis technologijomis grindžiamus, vis labiau kreipiančius daugiau dėmesio modeliavimui, o ne programavimui.
6. Pagrindiniai siūlomo sprendimo ypatumai yra šie:

a) sukurtas sprendimas minimizuoja žmogiškąjį klaidos faktorių, automatizuojant procesą, kuris paprastai atliekamas žmogaus rankomis;

b) paspartina kuriamų sistemų projektavimo etapą;

c) sprendimas sukurtas populiariausio UML įrankio MagicDraw aplinkoje, lengvai įdiegiamas ir naudojamas vieno mygtuko paspaudimu.

7. Sistemos realizacijai buvo panaudota MagicDraw UML įrankio aplinka.
8. Suprojektuotas ir realizuotas modelių transformavimo sprendimas leido praktiškai patikrinti darbe pasiūlytuosius teorinius sprendimus.

## Literatūra


[1] **Dean Leffingwell, Don Widrig.** Managing Software Requirements: A Use Case Approach, Second Edition. Addison Wesley, 2003, ISBN: 0-321-12247-X

[2] **Hans-Erik Eriksson and Magnus Penkeri.** Business Modeling With UML: Business Patterns at Work. John Wiley & Sons Inc., 2000, ISBN: 0471295515.

[3] **Tom Pender.** UML Bible. John Wiley & Sons Inc., 2000, ISBN: 0764526049.

[4] OMG Model Driven Architecture. [interaktyvus] 1997-2006 [žiūrėta 2006 06]. Prieiga per internetą: http://www.omg.org/mda/.

[5] Formal adoption of UML 2.0 superstructure. OMG RFP. Updated July 2005 [Žiūrėta 2006 06]. Prieiga per internetą: www.omg.org.

[6] **Martin Fowler.** UML Distilled: A Brief Guide to the Standard Object Modeling Language, Third Edition. Addison Wesley, 2003, ISBN: 0-321-19368-7.

[7] **Alistair Cockburn.** Writing effective Use Cases. Addison Wesley, 2000.

[8] **Pakalnickas E., Gudas S.** Sąsajomis grįstas IS projektavimas , konferencijoje "Informacinės technologijos – 2006", 2006.

[9] **David S. Frankel.** Model Driven Architecture: Applying MDA to Enterprise Computing. Wiley Publishing Inc., 2003.

[10] **Lopata A., Gudas S.** Veiklos modeliu grindžiamas vartotojo poreikių specifikavimas. KTU Informacijos sistemų katedra, VU KHF Informatikos katedra, 2005.

[11] **Tony Mallia.** Enterprise Change Methodology with MDA. CIBER Inc., 2001.

[12] **Desouza, D**. Model-Driven Architecture and Integration: Opportunities and Challenges. Version 1.1. 2001.

[13] **Cockburn, A.** Using Goal-Based Use Cases" JOOP vol. 10 no.7., 1997.

[14] **Coleman, D.** A Use Case Template: Draft for discussion, 1998.

## Terminų ir santrumpų žodynas

8. Lentelė. Terminai

| Sąvoka | Paaiškinimas |
|---|---|
| **IS** | Informacinė sistema |
| **PĮ** | Programinė įranga – bet kokia programinė įranga, kurią galima suprojektuoti |
| **CASE (Computer Aided Software Engineering)** | IS inžinerijos metodas – automatizuotas IS projektavimo procesas |
| **UML (unified modeling language)** | Unifikuota Modeliavimo Kalba – labiausiai naudojama specifikacija sistemų architektūros projektavime |
| **MDA (model driven architecture)** | Modeliavimu grįsta architektūra – architektūra, kurios pagalba PIM modelis paverčiamas PSM modeliu, t.y., pritaikomas kokiai nors platformai (XML, JAVA, CORBA) |
| **CIM (computation independent model)** | Neformalus modelis, kuris apibūdina modeliuojamos sistemos aplinką bei jai keliamus reikalavimus. |
| **PIM (platform independent model)** | Nepriklausomas nuo platformos modelis – modelis (ar posistemis), neturintis informacijos kokia sistema jis yra ar turi būti realizuotas |
| **PSM (platform specific model)** | Specifinis platformos modelis – modelis (ar posistemis), kuris pritaikytas realizacijai konkrečioje platformoje |
| **OMG (object management group)** | Atviras nariams ne pelno siekiantis konsorciumas, vystantis specifikacijas tarpusavyje bendraujančioms sistemoms. Geriausiai žinomos – MDA, UML, XMI ir kt. specifikacijos. |
| **UCM** | Use-case model – vartotojo reikalavimų modelis, nusakantis reikalavimus kuriamai sistemai, pagal numatyto verslo modelio automatizuojamus veiklos procesus. |

# Priedai

## 1 Priedas. Sprendimo realizacijos programos kodas.

XML failas "**plugin.xml**", kuris įkeliamas į MagicDraw įskiepių katalogą, kad būtų galima vykdyti sukurtą transformavimo algoritmą.

**Plugin.xml**

```
<?xml version="1.0" encoding="UTF-8" ?>
<plugin id="ActivityTransformPlugin" name="ActivityTransform" version="1.0" internalVersion="1" provider-
    name="KTU" class="activitytransform.ActivityTransformPlugin">
<runtime>
<library name="activitytransform.jar">
  <export name="*" />
    </library>
    </runtime>
    </plugin>
```

**ActivityDiagramTransformer.java**

```
/**
 * Copyright (c) 2007 NoMagic, Inc. All Rights Reserved.
 */
package activitytransform;

import com.nomagic.magicdraw.core.Project;
import com.nomagic.magicdraw.openapi.uml.ModelElementsManager;
import com.nomagic.magicdraw.openapi.uml.PresentationElementsManager;
import com.nomagic.magicdraw.openapi.uml.ReadOnlyElementException;
import com.nomagic.magicdraw.openapi.uml.SessionManager;
import com.nomagic.magicdraw.uml.symbols.DiagramPresentationElement;
import com.nomagic.magicdraw.uml.symbols.PresentationElement;
import com.nomagic.magicdraw.uml.symbols.shapes.SwimlaneHeaderView;
import com.nomagic.magicdraw.uml.symbols.shapes.SwimlaneView;
import com.nomagic.uml2.ext.magicdraw.activities.mdintermediateactivities.ActivityPartition;
import com.nomagic.uml2.ext.magicdraw.activities.mdfundamentalactivities.ActivityNode;
import com.nomagic.uml2.ext.magicdraw.classes.mdkernel.*;
import com.nomagic.uml2.ext.magicdraw.classes.mdkernel.Package;
import com.nomagic.uml2.impl.ElementsFactory;

import java.util.ArrayList;
import java.util.List;
import java.util.Collection;
import java.util.Iterator;

/**
 * @author
 */
public abstract class ActivityDiagramTransformer
{
        private ElementsFactory mElementsFactory;
        private Project mProject;

        private DiagramPresentationElement mActivityDiagramView;
        private DiagramPresentationElement mTargetDiagramView;

        private Diagram mDiagram;
        private Package mTargetPackage;

        public void init(DiagramPresentationElement diagramView)
        {
                mProject = Project.getProject(diagramView);
                mElementsFactory = mProject.getElementsFactory();
                mActivityDiagramView = diagramView;
                mDiagram = diagramView.getDiagram();
        }

        public void transform()
        {
                SessionManager sessionManager = SessionManager.getInstance();

                if (sessionManager.isSessionCreated())
                {
```

```
                sessionManager.closeSession();
            }

            sessionManager.createSession("Transform activity diagram");

            try
            {
                String targetName = mDiagram.getName() + " " + getTargetDiagramType();

                mTargetPackage = mElementsFactory.createPackageInstance();
                mTargetPackage.setOwner(mProject.getModel());
                mTargetPackage.setName(targetName);

                mTargetDiagramView = createTargetDiagram(mTargetPackage);
                mTargetDiagramView.getDiagram().setName(targetName);

                transformElements();
            }
            catch (ReadOnlyElementException e)
            {
                e.printStackTrace();
            }

            sessionManager.closeSession();

            mTargetDiagramView.open();
            mTargetDiagramView.layout(false);
        }

        private DiagramPresentationElement createTargetDiagram(Namespace owner)
        {
            return createDiagram(owner, getTargetDiagramType());
        }

        protected abstract String getTargetDiagramType();

        protected abstract PresentationElement createTargetForPartition(ActivityPartition partition)
throws
                ReadOnlyElementException;

        protected abstract PresentationElement createTargetForPartitionActivityNode(ActivityNode
activityNode)
                throws ReadOnlyElementException;

        protected abstract PresentationElement
associatePartitionAndActivityNodeTargets(PresentationElement partitionTarget,
                                                PresentationElement activityNodeTarget)
                                                throws ReadOnlyElementException;

        private void transformElements() throws ReadOnlyElementException
        {
            prepareToAddTargetsForPartitionNodes();

            List partitions = getAllPartitions();

            for (int i = 0; i < partitions.size(); i++)
            {
                ActivityPartition partition = (ActivityPartition) partitions.get(i);

                PresentationElement partitionTarget = createTargetForPartition(partition);
                createTargetsForPartitionNodes(partition, partitionTarget);
            }
        }

        private void createTargetsForPartitionNodes(ActivityPartition partition, PresentationElement
partitionTarget)
                throws ReadOnlyElementException
        {
            if (partition.hasContainedNode())
            {
                Collection containedNodes = partition.getContainedNode();

                for (Iterator iterator = containedNodes.iterator(); iterator.hasNext();)
                {
                    ActivityNode node = (ActivityNode) iterator.next();
                    PresentationElement activityNodeTarget =
createTargetForPartitionActivityNode(node);
                    associatePartitionAndActivityNodeTargets(partitionTarget,
activityNodeTarget);
```

```
                        }
                }
        }

        protected void prepareToAddTargetsForPartitionNodes() throws ReadOnlyElementException
        {

        }

        protected Namespace getTargetDataParent(DiagramPresentationElement diagramView)
        {
                return mTargetPackage;
        }

        private DiagramPresentationElement createDiagram(Namespace owner, String type)
        {
                DiagramPresentationElement diagramView = null;

                try
                {
                        Diagram diagram = ModelElementsManager.getInstance().createDiagram(type, owner);

                        diagramView = mProject.getDiagram(diagram);
                }
                catch (ReadOnlyElementException e)
                {
                        e.printStackTrace();
                }

                return diagramView;
        }

        private List getAllPartitions()
        {
                List partitions = new ArrayList();
                List views = mActivityDiagramView.getPresentationElements();

                for (int i = 0; i < views.size(); i++)
                {
                        PresentationElement view = (PresentationElement) views.get(i);

                        if (view instanceof SwimlaneView)
                        {
                                SwimlaneView swimlaneView = (SwimlaneView) view;
                                List headers = swimlaneView.getAllSwimlanes();

                                for (int j = 0; j < headers.size(); j++)
                                {
                                        SwimlaneHeaderView headerView = (SwimlaneHeaderView)
headers.get(j);
                                        partitions.add(headerView.getElement());
                                }
                        }
                }

                return partitions;
        }

        protected ElementsFactory getElementsFactory()
        {
                return mElementsFactory;
        }

        protected Project getProject()
        {
                return mProject;
        }

        public DiagramPresentationElement getActivityDiagramView()
        {
                return mActivityDiagramView;
        }

        public DiagramPresentationElement getTargetDiagramView()
        {
                return mTargetDiagramView;
        }

        public Package getTargetPackage()
```

```
	{
		return mTargetPackage;
	}
}
```

**ActivityTransformPlugin.java**

```
/**
 * Copyright (c) 2007 NoMagic, Inc. All Rights Reserved.
 */
package activitytransform;

import com.nomagic.actions.AMConfigurator;
import com.nomagic.actions.ActionsManager;
import com.nomagic.actions.NMAction;
import com.nomagic.magicdraw.actions.*;
import com.nomagic.magicdraw.core.Application;
import com.nomagic.magicdraw.core.Project;
import com.nomagic.magicdraw.plugins.Plugin;
import com.nomagic.magicdraw.uml.DiagramType;
import com.nomagic.magicdraw.uml.symbols.DiagramPresentationElement;

import javax.swing.*;
import java.awt.event.ActionEvent;
import java.awt.event.KeyEvent;
import java.awt.*;

/**
 * @author
 */
public class ActivityTransformPlugin extends Plugin
{
	public void init()
	{
		System.out.println("ActivityTransformPlugin.init");

		configureMainMenu();
	}

	private static void configureMainMenu()
	{
		ActionsConfiguratorsManager.getInstance().addMainMenuConfigurator(new AMConfigurator()
		{
			public void configure(ActionsManager mngr)
			{
				MDActionsCategory actionsGroup = new
MDActionsCategory("ACTIVITY_DIAGRAM_TRANSFORMATION",

				   "Activity Diagram Transformation");

				actionsGroup.setNested(true);
				actionsGroup.addAction(createTransformAction());

				NMAction action = mngr.getActionFor(ActionsID.TOOLS_PLUGINS);
				if (action != null)
				{
					action.addAction(actionsGroup);
				}
			}

			public int getPriority()
			{
				return AMConfigurator.MEDIUM_PRIORITY;
			}
		});
	}

	private static MDAction createTransformAction()
	{
		return new MDAction("TRANSFORM_ACTIVE_DIAGRAM", "Transform Active Diagram",
				KeyStroke.getKeyStroke(KeyEvent.VK_T,
Toolkit.getDefaultToolkit().getMenuShortcutKeyMask()),
					ActionsGroups.DIAGRAM_OPENED_RELATED)
		{
			public void actionPerformed(ActionEvent actionEvent)
			{
				DiagramPresentationElement activeDiagram = getActiveActivityDiagram();
```

```
                                ActivityDiagramTransformer transformer = new
ActivityToUseCaseDiagramTransformer();
                                transformer.init(activeDiagram);
                                transformer.transform();

//                              transformer = new ActivityToStateDiagramTransformer();
//                              transformer.init(activeDiagram);
//                              transformer.transform();
//
//                              transformer = new ActivityToSequenceDiagramTransformer();
//                              transformer.init(activeDiagram);
//                              transformer.transform();
                        }

                        public void updateState()
                        {
                                setEnabled(getActiveActivityDiagram() != null);
                        }
                };
        }

        private static DiagramPresentationElement getActiveActivityDiagram()
        {
                Project project = Application.getInstance().getProject();

                if (project != null)
                {
                        DiagramPresentationElement activeDiagram = project.getActiveDiagram();

                        if (activeDiagram != null)
                        {
                                DiagramType diagramType = activeDiagram.getDiagramType();

                                if (diagramType.isTypeOf(DiagramType.UML_ACTIVITY_DIAGRAM))
                                {
                                        return activeDiagram;
                                }
                        }
                }

                return null;
        }

        public boolean close()
        {
                return true;
        }

        public boolean isSupported()
        {
                return true;
        }
}
```

**ActivityToUseCaseDiagramTransformer.java**

```
/**
 * Copyright (c) 2007 NoMagic, Inc. All Rights Reserved.
 */
package activitytransform;

import com.nomagic.magicdraw.openapi.uml.PresentationElementsManager;
import com.nomagic.magicdraw.openapi.uml.ReadOnlyElementException;
import com.nomagic.magicdraw.uml.DiagramTypeConstants;
import com.nomagic.magicdraw.uml.symbols.PresentationElement;
import com.nomagic.magicdraw.uml.symbols.shapes.ShapeElement;
import com.nomagic.uml2.UML2Constants;
import com.nomagic.uml2.ext.jmi.helpers.StereotypesHelper;
import com.nomagic.uml2.ext.jmi.helpers.ModelHelper;
import com.nomagic.uml2.ext.magicdraw.activities.mdfundamentalactivities.ActivityNode;
import com.nomagic.uml2.ext.magicdraw.activities.mdintermediateactivities.ActivityPartition;
import com.nomagic.uml2.ext.magicdraw.classes.mdkernel.Association;
import com.nomagic.uml2.ext.magicdraw.mdusecases.Actor;
import com.nomagic.uml2.ext.magicdraw.mdusecases.UseCase;

/**
 * @author
 */
public class ActivityToUseCaseDiagramTransformer extends ActivityDiagramTransformer
{
```

```
        private ShapeElement mSystem;

        protected String getTargetDiagramType()
        {
                return DiagramTypeConstants.UML_USECASE_DIAGRAM;
        }

        protected PresentationElement createTargetForPartition(ActivityPartition partition) throws
ReadOnlyElementException
        {
                Actor actor = getElementsFactory().createActorInstance();

                actor.setName(partition.getName());
                actor.setOwner(getTargetPackage());

                return PresentationElementsManager.getInstance().createShapeElement(actor,
getTargetDiagramView());
        }

        protected PresentationElement createTargetForPartitionActivityNode(ActivityNode activityNode)
throws ReadOnlyElementException
        {
                UseCase useCase = getElementsFactory().createUseCaseInstance();

                useCase.setName(activityNode.getName());
                useCase.setOwner(mSystem.getElement());

                return PresentationElementsManager.getInstance().createShapeElement(useCase, mSystem);
        }

        protected void prepareToAddTargetsForPartitionNodes() throws ReadOnlyElementException
        {
                com.nomagic.uml2.ext.magicdraw.classes.mdkernel.Package system =
getElementsFactory().createPackageInstance();
                system.setName("System");
                system.setOwner(getTargetPackage());
                StereotypesHelper.addStereotype(system, StereotypesHelper.getStereotype(getProject(),

                                        UML2Constants.UML_USECASEMODEL_STEREOTYPE));

                mSystem = PresentationElementsManager.getInstance().createShapeElement(system,
getTargetDiagramView());
        }

        protected PresentationElement associatePartitionAndActivityNodeTargets(PresentationElement
partitionTarget,
                                                        PresentationElement activityNodeTarget)
                                                        throws ReadOnlyElementException
        {
                Association association = getElementsFactory().createAssociationInstance();
                association.setOwner(getTargetPackage());

                ModelHelper.setClientElement(association, partitionTarget.getElement());
                ModelHelper.setSupplierElement(association, activityNodeTarget.getElement());

                ModelHelper.setNavigable(ModelHelper.getFirstMemberEnd(association), true);
                ModelHelper.setNavigable(ModelHelper.getSecondMemberEnd(association), true);

                return PresentationElementsManager.getInstance().createPathElement(association,
partitionTarget,

                                        activityNodeTarget);
        }
}
```

**ActivityToStateDiagramTransformer.java**

```
/**
 * Copyright (c) 2007 NoMagic, Inc. All Rights Reserved.
 */
package activitytransform;

import com.nomagic.magicdraw.uml.symbols.shapes.ShapeElement;
import com.nomagic.magicdraw.uml.symbols.PresentationElement;
import com.nomagic.magicdraw.uml.symbols.DiagramPresentationElement;
import com.nomagic.magicdraw.uml.DiagramTypeConstants;
import com.nomagic.magicdraw.openapi.uml.ReadOnlyElementException;
import com.nomagic.magicdraw.openapi.uml.PresentationElementsManager;
import com.nomagic.uml2.ext.magicdraw.activities.mdintermediateactivities.ActivityPartition;
import com.nomagic.uml2.ext.magicdraw.activities.mdfundamentalactivities.ActivityNode;
```

```
import com.nomagic.uml2.ext.magicdraw.mdusecases.Actor;
import com.nomagic.uml2.ext.magicdraw.mdusecases.UseCase;
import com.nomagic.uml2.ext.magicdraw.classes.mdkernel.Association;
import com.nomagic.uml2.ext.magicdraw.classes.mdkernel.Namespace;
import com.nomagic.uml2.ext.magicdraw.statemachines.mdbehaviorstatemachines.State;
import com.nomagic.uml2.ext.jmi.helpers.StereotypesHelper;
import com.nomagic.uml2.ext.jmi.helpers.ModelHelper;
import com.nomagic.uml2.UML2Constants;

/**
 * @author
 */
public class ActivityToStateDiagramTransformer extends ActivityDiagramTransformer
{
	protected String getTargetDiagramType()
	{
		return DiagramTypeConstants.UML_STATECHART_DIAGRAM;
	}

	protected PresentationElement createTargetForPartition(ActivityPartition partition) throws
ReadOnlyElementException
	{
		return null;
	}

	protected PresentationElement createTargetForPartitionActivityNode(ActivityNode activityNode)
throws ReadOnlyElementException
	{
		State state = getElementsFactory().createStateInstance();

		state.setName(activityNode.getName());
		state.setOwner(getTargetDataParent(getTargetDiagramView()));

		return PresentationElementsManager.getInstance().createShapeElement(state,
getTargetDiagramView());
	}

	protected void prepareToAddTargetsForPartitionNodes() throws ReadOnlyElementException
	{

	}

	protected PresentationElement associatePartitionAndActivityNodeTargets(PresentationElement
partitionTarget,
							PresentationElement activityNodeTarget)
							throws ReadOnlyElementException
	{
		return null;
	}

	protected Namespace getTargetDataParent(DiagramPresentationElement diagramView)
	{
		return (Namespace) diagramView.getDiagram().getOwner();
	}
}
```

**ActivityToSequenceDiagramTransformer.java**

```
/**
 * Copyright (c) 2007 NoMagic, Inc. All Rights Reserved.
 */
package activitytransform;

import com.nomagic.magicdraw.uml.DiagramTypeConstants;
import com.nomagic.magicdraw.uml.symbols.DiagramPresentationElement;
import com.nomagic.magicdraw.uml.symbols.PresentationElement;
import com.nomagic.magicdraw.uml.symbols.shapes.ShapeElement;
import com.nomagic.magicdraw.uml.symbols.shapes.LifelineView;
import com.nomagic.magicdraw.openapi.uml.ReadOnlyElementException;
import com.nomagic.magicdraw.openapi.uml.PresentationElementsManager;
import com.nomagic.uml2.ext.magicdraw.activities.mdintermediateactivities.ActivityPartition;
import com.nomagic.uml2.ext.magicdraw.activities.mdfundamentalactivities.ActivityNode;
import com.nomagic.uml2.ext.magicdraw.activities.mdbasicactivities.ActivityEdge;
import com.nomagic.uml2.ext.magicdraw.classes.mdkernel.Namespace;
import com.nomagic.uml2.ext.magicdraw.interactions.mdbasicinteractions.Lifeline;

import java.awt.*;
import java.util.Map;
import java.util.HashMap;
import java.util.Collection;
```

```
import java.util.Iterator;

/**
 * @author
 */
public class ActivityToSequenceDiagramTransformer extends ActivityDiagramTransformer
{
	private Map mPartition2LifelineMap = new HashMap();

	protected String getTargetDiagramType()
	{
		return DiagramTypeConstants.UML_SEQUENCE_DIAGRAM;
	}

	protected PresentationElement createTargetForPartition(ActivityPartition partition) throws
			ReadOnlyElementException
	{
		Lifeline lifeline = getElementsFactory().createLifelineInstance();

		lifeline.setName(partition.getName());
		lifeline.setOwner(getTargetDataParent(getTargetDiagramView()));

		ShapeElement shape = PresentationElementsManager.getInstance()
				.createShapeElement(lifeline, getTargetDiagramView());

		mPartition2LifelineMap.put(partition, shape);

		moveLifelineShapeElement(shape);

		return shape;
	}

	private static void moveLifelineShapeElement(ShapeElement element) throws
ReadOnlyElementException
	{
		Rectangle bounds = element.getBounds();
		bounds.x += 50;

		PresentationElementsManager.getInstance().reshapeShapeElement(element, bounds);
	}

	protected Namespace getTargetDataParent(DiagramPresentationElement diagramView)
	{
		return (Namespace) diagramView.getDiagram().getOwner();
	}

	protected PresentationElement createTargetForPartitionActivityNode(ActivityNode activityNode)
	{
		if (activityNode.hasInPartition())
		{
			ActivityPartition partition = (ActivityPartition)
activityNode.getInPartition().iterator().next();

			if (activityNode.hasOutgoing())
			{
				Collection outgoing = activityNode.getOutgoing();

				for (Iterator iterator = outgoing.iterator(); iterator.hasNext();)
				{
					ActivityEdge edge = (ActivityEdge) iterator.next();
					ActivityNode targetNode = (ActivityNode) edge.getTarget();

					if(targetNode.hasInPartition())
					{
						ActivityPartition targetPartition = (ActivityPartition)
targetNode.getInPartition().iterator().next();
						createMessage(partition, targetPartition);
					}
				}
			}
		}

		return null;
	}

	private void createMessage(ActivityPartition source, ActivityPartition target)
	{

	}
```

```
	protected PresentationElement associatePartitionAndActivityNodeTargets(PresentationElement
partitionTarget,

                                        PresentationElement activityNodeTarget)
                        throws ReadOnlyElementException
	{
		return null;
	}
}
```

## 2 Priedas. Sprendimo įskiepio diegimo į MagicDraw instrukcija

Norint, kad įrankio Magic Draw aplinkoje veiktų sukurtas įskiepis, sukompiliuotą programos kodą, esantį dviejuose failuose (activitytransform.jar ir plugin.xml) su visu katalogu nukopijuoti į MagicDraw įskiepių katalogą. Standartiniu atveju, šis kelias yra: c:\Program Files\MagicDraw UML\plugins\. Iš naujo paleidus programą, įskiepis aktyvuojamas ir paruoštas naudojimui.