KAUNO TECHNOLOGIJOS UNIVERSITETAS

INFORMATIKOS FAKULTETAS

Ieva Gaidukevičiūtė

ICONIX METODU KURIAMO *UML* MODELIO ELEMENTŲ SUDERINAMUMO TYRIMAS

Baigiamasis magistro projektas

Vadovas
doc. dr. L. Čeponienė

KAUNAS, 2017

KAUNO TECHNOLOGIJOS UNIVERSITETAS

INFORMATIKOS FAKULTETAS

ICONIX METODU KURIAMO *UML* MODELIO ELEMENTŲ SUDERINAMUMO TYRIMAS

Baigiamasis magistro projektas
Informacinių sistemų inžinerijos studijų programa (kodas 621E15001)

Vadovas
doc. dr. L. Čeponienė
2017-05-22

Recenzentas
lekt. dr. M. Binkis
2017-05-22

Projektą atliko
Ieva Gaidukevičiūtė
2017-05-22

KAUNAS, 2017

KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS

(Fakultetas)

Ieva Gaidukevičiūtė

(Studento vardas, pavardė)

Informacinių sistemų inžinerijos studijų programa, 621E15001

(Studijų programos pavadinimas, kodas)

Baigiamojo projekto
„*ICONIX* metodu kuriamo *UML* modelio elementų suderinamumo tyrimas“

AKADEMINIO SĄŽININGUMO DEKLARACIJA

20 17 m. gegužės 22 d.
Kaunas

Patvirtinu, kad mano, **Ievos Gaidukevičiūtės**, baigiamasis projektas tema „*ICONIX* metodu kuriamo *UML* modelio elementų suderinamumo tyrimas“ yra parašytas visiškai savarankiškai ir visi pateikti duomenys ar tyrimų rezultatai yra teisingi ir gauti sąžiningai. Šiame darbe nei viena dalis nėra plagijuota nuo jokių spausdintinių ar internetinių šaltinių, visos kitų šaltinių tiesioginės ir netiesioginės citatos nurodytos literatūros nuorodose. Įstatymų nenumatytų piniginių sumų už šį darbą niekam nesu mokėjęs.

Aš suprantu, kad išaiškėjus nesąžiningumo faktui, man bus taikomos nuobaudos, remiantis Kauno technologijos universitete galiojančia tvarka.

(vardą ir pavardę įrašyti ranka) *(parašas)*

Gaidukevičiūtė, Ieva. *ICONIX* metodu kuriamo *UML* modelio elementų suderinamumo tyrimas. Magistro baigiamasis projektas / vadovas doc. dr. Lina Čeponienė; Kauno technologijos universitetas, Informatikos fakultetas.

Mokslo kryptis ir sritis: Informatikos inžinerija, technologijos mokslai

Reikšminiai žodžiai: *ICONIX, UML, MagicDraw, OCL, suderinamumas.*

Kaunas, 2017. 88 p.

SANTRAUKA

UML (angl. *Unified Modeling Language*) - viena dažniausiai naudojamų sistemų modeliavimo kalbų. Paprastai skirtingos UML diagramos tarpusavyje yra susijusios ir vienų diagramų elementai yra naudojami kitose. UML įrankiuose paliekama daug laisvės ir dažnai galima sukurti nekorektiškų diagramų, kai viena diagrama su tokiu pačiu elementu rodo vienokį elgesį, o kita – visiškai kitokį. Rankiniu būdu suderinti UML modelio diagramas ir jų elementus tarpusavyje yra pakankamai sudėtinga ir toks suderinimas reikalauja daug pastangų. Tokio elementų tarpusavio suderinimo proceso dalinis automatizavimas palengvintų darbą.

Darbe pateiktas siūlymas problemai spręsti – sukurtas taisyklių rinkinys. Atsižvelgiant į tai, kad UML yra tik kalba, kuri nusako, kaip modeliuoti sistemas, o kuriant sistemą dažniausiai vadovaujamasi procesu, pagal kurį kuriamos diagramos, buvo pasirinktas metodas. Sistemų kūrimo metodų yra labai daug ir įvairių. Vienas iš tokių yra ICONIX procesas, kuris remiasi UML. Remiantis UML diagramomis ir jų elementais, sukurtas taisyklių rinkinys užtikrina tiek pagrindinių UML naudojamų elementų ir diagramų tarpusavio ryšius, jų korektiškumą, tiek pasirinkto ICONIX metodo pagrindinius principus skirtinguose modelio kūrimo etapuose. Realizacijai pasirinkta *MagicDraw* programa, į kurią perkėlus darbe aprašytas ir sukurtas taisykles patikrinami projektų, kurtų pagal ICONIX metodą, diagramų tarpusavio ryšiai, jų teisingumas ir korektiškumas. Sukurto taisyklių rinkinio veikimas išbandytas patobulintame įrankyje ir eksperimentiškai ištirtas su trimis skirtingais projektų variantais.

Sukurtas taisyklių rinkinys palengvina pagrindinių ICONIX principų taikymą ir UML diagramų elementų tarpusavio suderinimą. Taisyklių rinkinys gali būti taikomas skirtinguose ICONIX proceso etapuose taip palengvinant sistemos projektuotojo darbą, kai kuriamas sistemos modelis.

Gaidukevičiūtė, Ieva. *Study of ICONIX Method-Based UML Model Elements' Consistency*: Master‘s thesis in Information Systems Engineering / supervisor assoc. prof. Lina Čeponienė. The Faculty of Informatics, Kaunas University of Technology.

Research area and field: Informatics Engineering, Technology Science

Key words: *ICONIX, UML, MagicDraw, OCL, Consistency.*

Kaunas, 2017. 88 p.

SUMMARY

UML (Unified Modeling Language) is one of the most often used languages in system modeling. Typically, different UML diagrams are related and elements of some diagrams are used in other diagrams. There are plenty of choices how to use elements in diagrams while working on UML tools and it is possible to create incorrect diagrams often, where one diagram indicates one kind of behavior and other diagram indicates totally different behavior. It is hard to manually coordinate UML model diagrams and their elements and this consistency requires a lot of effort. Partial automatization of process of elements consistency would facilitate the work.

The thesis presents the suggestion to solve the problem. The set of rules is created which ensures the connections and correctness between main elements in UML and diagrams as well as main principles of ICONIX method in different stages of model creating. *MagicDraw* program was chosen for realization, where the rules, created and defined in this thesis, enables to check the connections of diagrams, regularity and correctness of the projects, created by ICONIX method. The operation of set of rules was checked on enhanced tool and experimentally tested with three different versions of the projects.

The set of rules facilitates the usage of main ICONIX principles and consistency of UML diagrams elements. The collection of rules may be applied in different stages of ICONX process in order to facilitate the work of system designer while creating the system model.

TURINYS

LENTELIŲ SĄRAŠAS

PAVEIKSLŲ SĄRAŠAS

TERMINŲ IR SANTRUMPŲ ŽODYNAS

Santrumpa, Terminas	Paaiškinimas
UML *(*angl. *Unified Modeling Language)*	Unifikuota modeliavimo kalba, skirta programinei įrangai ir sistemoms modeliuoti.
ICONIX	Programinių ir informacinių sistemų kūrimo metodas, aprašantis, kaip modeliuoti ir kurti sistemas.
Robustness diagram	Išbaigtumo diagrama - viena iš UML naudojančių diagramų, pritaikyta ICONIX procesui modeliuoti.
OCL (angl. *Object Constraint Language*)	Formali kalba, skirta UML modelių išraiškoms aprašyti.

ĮVADAS

UML (angl. *Unified Modeling Language*) – tai standartas, skirtas modeliuoti tiek programinės įrangos, tiek kitokias sistemas. Tai viena dažniausiai programinės įrangos kūrimo įrankiuose naudojamų kalbų. UML standartu galima aprašyti sistemos reikalavimus, apimančius tokius konceptualius dalykus, kaip verslo procesai ar sistemos funkcijos, taip pat tokius konkrečius dalykus, kaip klasės, duomenų bazių schemos ir pan. Modeliuojant sistemą skirtingos UML diagramos, perteikiančios sistemą iš skirtingų požiūrio taškų, tarpusavyje yra susijusios ir vienų diagramų elementai yra naudojami kitose. UML įrankiuose paliekama daug laisvės ir dažnai galima sukurti nekorektiškų diagramų, kai viena diagrama su tokiu pačiu elementu rodo vienokį elgesį, o kita – visiškai kitokį. Taip sukuriant kelis elementus, kurie turi tokią pačią reikšmę sistemoje, diagramų elementai tarpusavyje tampa nesuderinti, kai gali būti perpanaudotas tas pats elementas keliose skirtingose diagramose.

Įvairūs skirtingi sistemų modeliai nuolatos kuriami įvairiose programose, tačiau įrankiuose paliekama nemažai laisvės sudarant skirtingas diagramas, todėl didėja tikimybė palikti modelyje klaidų, trūkumų. Rankiniu būdu suderinti UML modelio diagramas ir jų elementus tarpusavyje yra pakankamai sudėtinga ir toks suderinimas reikalauja daug pastangų. Tokio elementų tarpusavio suderinimo proceso dalinis automatizavimas palengvintų darbą. Kuriantiems modelius sudėtinga stebėti visus elementus, kaip jie naudojami diagramose, o procesą bent dalinai automatizavus – kūrimo procesas ne tik palengvėtų, bet ir pagreitėtų. UML nenurodo, kaip reikėtų kurti diagramas, ši kalba tik aprašo skirtingus elementus, naudojamus sistemoms modeliuoti. UML neturi suderinamumo užtikrinimo funkcijos, tačiau *MagicDraw* programoje, naudojančioje UML, yra verifikavimo įrankis. Naudojantis šiuo įrankiu programoje galima patikrinti tokius bendrinius reikalavimus, kaip ar elementas turi pavadinimą, ar klasės atributas turi pavadinimą, jeigu jam nurodytas tipas, ir panašiai, tačiau šis įrankis neužtikrina modelių vientisumo, jeigu modelis kuriamas naudojant specifinį programinės įrangos kūrimo metodą. Šiame darbe nagrinėjamas ICONIX metodas, kuriame aprašytos tik jam būdingos informacinės sistemos kūrimo taisyklės, tokios kaip apribojimai išbaigtumo diagramų elementams ir ryšiams ar specifinių elementų naudojimas. Šiuo metodu kuriamo modelio elementų suderinamumą sudėtinga patikrinti rankiniu būdu, o *MagicDraw* programa neužtikrina metodo specifinių taisyklių patikrinimo.

Šio darbo **tikslas** yra palengvinti UML modelio kūrimą ir pagerinti modelio kokybę pasiūlant modelio elementų tarpusavio suderinimo priemones ir įrankius. Šis tikslas buvo pasiektas atlikus išsikeltus darbui **uždavinius**:

1. Išanalizuoti UML kalbą ir diagramų tarpusavio ryšius.
2. Išanalizuoti metodus, naudojančius UML, pasirinkti metodą, kurio metu kuriamas diagramas būtų naudinga suderinti.
3. Išanalizuoti suderinimo priemones ir pasirinkti tinkamą.
4. Išanalizuoti įrankius, kuriuose būtų galima taikyti suderinimo priemones, ir pasirinkti tinkamą įrankį.
5. Pasiūlyti metodiką UML modelio suderinimui.
6. Pasiūlytą metodiką realizuoti pasirinktame įrankyje.
7. Išbandyti metodikos veikimą patobulintame įrankyje.
8. Eksperimentiškai ištirti metodikos taikymą praktikoje.

Išanalizavus UML kalbą, diagramų tarpusavio ryšius ir metodus, naudojančius UML kalbą, pasirinktas ICONIX metodas, kurio metu kuriamas diagramas būtų naudinga suderinti. Norint kuriamą sistemą modeliuoti UML kalba, reikia pasirinkti programą, kurioje galėtų būti atliekamas sistemos projektavimas. Sukurta nemažai skirtingų įrankių, kurie yra pritaikyti UML modeliuoti ir kurti. Įrankiai palaiko įvairias UML versijas, tokias kaip 2.1. 2.2 ar 2.5 ir kt. Vienas iš tokių įrankių yra *MagicDraw* programa, kuri palaiko 2.5 UML versiją. Ši programa turi galimybes modelius kurti ne tik UML, bet ir kitomis kalbomis, taip pat programoje galima taikyti skirtingus metodus kuriant informacinių sistemų projektus. Taip pat ši programa turi integruotą modulį, kuris užtikrina kuriamų

modelių korektiškumą, nes galima patikrinti modelio suderinamumą pritaikant validacijos taisykles, įtrauktas į programą pagal nutylėjimą. Esamam validacijos įrankiui pasiūlytas įrankio papildymas taisyklėmis, kurios tikrina elementų suderinamumą taikant ICONIX metodą. Pasiūlytos taisyklės UML modelio suderinimui realizuotos *MagicDraw* įrankyje. *MagicDraw* programos validacijos modulyje yra galimybė sukurti ir įtraukti naujas taisykles, kurios gali būti užrašomos ir OCL (angl. *Object Constraint Language*). OCL yra formali kalba, skirta UML modelių išraiškoms aprašyti. Paprastai šios išraiškos aprašo nekintamas sąlygas, kol sistema modeliuojama. Taigi naudojant OCL patogu užrašyti taisykles, pritaikytas ICONIX metodu kuriamam modeliui, reikalingas įtraukti į *MagicDraw* įrankį.

Realizavus tiek UML, tiek ICONIX metodui pritaikytas taisykles, sukurto taisyklių rinkinio veikimas išbandytas patobulintame įrankyje ir eksperimentiškai ištirtas praktikoje. Šiame darbe sukurtas ir pasiūlytas taisyklių rinkinys ištirtas eksperimentiškai, kai taisyklių rinkinio veikimas pritaikytas trimis etapais. Pirmu etapu taisyklių rinkinys buvo pritaikytas projektui, aprašytame D.Rosenberg‘o ir K.Scott‘o knygoje „Applying use case driven object modeling with UML“ [1]. Taip pat šioje knygoje aprašytas ir darbe taikomas ICONIX metodas. Taip pat buvo sukurtas naujas projektas taikant ICONIX metodą. Tiriant taisyklių veikimą trečiajame etape buvo surinkti trys skirtingi studentų praktiniai darbai, atlikti viename iš universitete vedamų modulių, kuriame mokoma kurti informacines sistemas. Visais trimis tyrimo etapais buvo suskaičiuota, kiek diagramų ir elementų sudaro kiekvieną projektą, tada sukurtiems projektams buvo pritaikyti taisyklių rinkiniai, ir sekama, kiek klaidų aptinkama.

Darbo rezultatas yra sukurtas ir pasiūlytas taisyklių rinkinys UML modelio elementų suderinimui, o pats taisyklių rinkinio veikimas išbandytas patobulintame įrankyje ir eksperimentiškai ištirtas taisyklių rinkinio taikymas praktikoje. Šie darbo rezultatai svarbūs tuo, kad sukurtas taisyklių rinkinys, tikrinantis ICONIX metodu kurto UML modelio diagramų elementų suderinamumą tarpusavyje, sumažina tikimybę ir galimybes kurti nelogiškas ar tarpusavyje nesuderintas diagramas. Taip pat šis diagramų kūrimo proceso dalinis automatizavimas palengvina darbą, nebereikia tiek daug pastangų bandant rankiniu būdu suderinti UML modelio diagramas ar diagramų elementus.

Darbą sudaro šeši skyriai, ir trys priedai. Darbo apimtis – 88 puslapiai. Pirmajame skyriuje analizuojama UML kalba, jos taikymas, UML diagramos ir jų tarpusavio ryšiai. Taip pat analizuojami UML metamodeliai, kuriais remiantis kuriamos validacijos taisyklės. Toliau analizuojamas ICONIX metodas, skirtas informacinėms sistemoms modeliuoti. Išanalizuojama, kaip suderinimo įrankis veikia *MagicDraw* programoje, kas jau yra realizuota, kaip būtų galima įtraukti naujas suderinimo taisykles ir paketus. Pateikiama su darbo tema susijusių šaltinių analizė, palyginti penki panašūs sprendimai, susiję su UML diagramų suderinimu. Antrajame skyriuje pateikiamas problemos sprendimo aprašas. Aprašoma, kaip galima sukurti ir įtraukti naują validacijos taisyklę į *MagicDraw* įrankį, kad ją būtų galima naudoti kuriamiems modeliams tikrinti. Išgryninamos taisyklės, reikalingos ICONIX metodu kuriamo modelio suderinimui užtikrinti. Trečiajame skyriuje aprašoma kaip sprendimas veikia, jo realizacija. Natūralia kalba aprašytos suderinimo taisyklės, pritaikytos ICONIX metodu kuriamam modeliui tikrinti, užrašomos OCL. Taip pat pateikiamos rekomendacijos vėliau taisykles naudojančiam vartotojui, kaip kurti projektą, kad kuriamam projektui būtų galima lengviau pritaikyti sukurtas validacijos taisykles. Ketvirtajame skyriuje aprašomas eksperimentinis sprendimo taikymas. Eksperimento tikslas - patikrinti sukurtų validacijos taisyklių veikimą *MagicDraw* projektuose, kurtuose taikant ICONIX metodą. Eksperimente pateikiami trys būdai, kaip ištirtas taisyklių rinkinio veikimas skirtinguose projektuose. Taip pat pateikiamas ekperimento įvertinimas – validacijos taisyklės tikrina skirtingus, dažniausiai ICONIX metode naudojamus metamodelio elementus, o pritaikius validacijos taisykles užtikrinamas šių pagrindinių elementų suderinamumas. Taip pat pateikiamos tokios atlikto eksperimento rekomendacijos, kaip kuriamame projekte reikėtų taikyti aplankų struktūrą, kurti tikslias diagramas, tada būtų galima užtikrinti lengvesnį ir patogesnį taisyklių taikymą projektui. Darbas baigiamas išvadomis, kurios yra pateikiamos penktame skyriuje.

1. PROBLEMINĖS SRITIES ANALIZĖ

1.1. Analizės tikslas

Šio darbo analizės tikslas – išanalizavus kuo daugiau informacijos šaltinių darbo rašymo tema ir išsirinkus tinkamiausią metodą, naudojantį UML, taip pat suderinimo priemonę, tikrinančią diagramas ir jų ryšius, ir įrankį, kuriame taikoma suderinimo priemonė, pasiūlomas UML modelio diagramų ir jų elementų suderinimo sprendimas, kai tikrinama, ar persidengiantys elementai skirtinguose diagramų modeliuose atitinka vienas kitą.

1.2. Tyrimo objektas, sritis ir problema

Šio darbo tyrimo objektas - UML modelio suderinimas. Įrankiui palikus daug laisvės, kuriant projektą kartais gali būti sukuriamos nelogiškos diagramos, kai tas pats elementas, turintis būti keliose diagramose ir vaizduojantis tą pačią sistemos dalį, skirtingose diagramose sukuriamas kaip atskiras elementas, nors turėtų būti perpanaudotas tas pats elementas keliose diagramose.

Šio darbo tyrimo sritis - UML modeliavimas pagal pasirinktą ICONIX metodą.

Tyrimo problematika - skirtingos UML diagramos yra tarpusavyje susiję ir elementai iš vienų diagramų yra naudojami kitose. UML įrankiuose paliekama daug laisvės, ir dažnai galima sukurti nelogiškų diagramų, kai viena diagrama su tokiu pačiu elementu rodo vienokį elgesį, o kita - visiškai kitokį, ir tada elementai tarpusavyje nedera. Rankiniu būdu suderinti UML modelio diagramas ir jų elementus tarpusavyje pakankamai sudėtinga ir reikalauja daug pastangų. Tokio proceso dalinis automatizavimas palengvintų darbą.

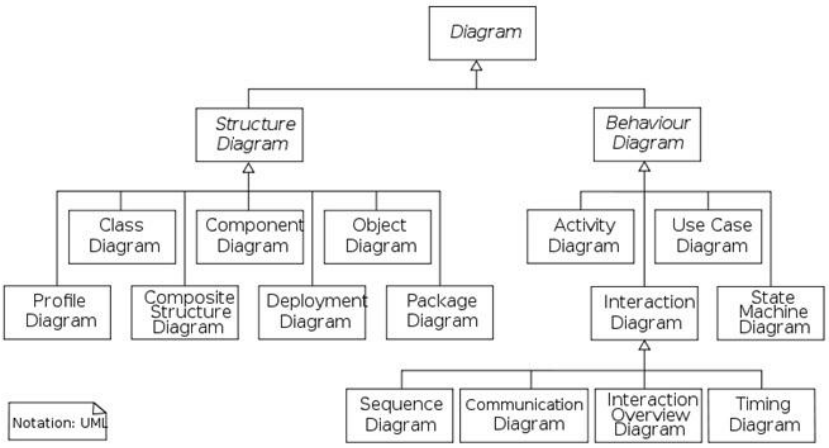
1.2.1. UML, UML taikymas ir UML diagramos

UML yra standartas, skirtas modeliuoti sistemas, ypatingai programinės įrangos. UML kalba buvo sukurta siekiant sumažinti arba visiškai pašalinti potencialius bereikalingus skirtingų metodų skirtumus, kurie vartotojus lengvai supainioja. Kalba buvo kuriama sujungiant skirtingus jau sukurtus metodus, taip užtikrinamas objektinės rinkos stabilumas, skatinantis projektus nusistovėti vienoje brandžioje modeliavimo kalboje, o kūrėjai įgalinami sutelkti dėmesį kuriant daugiau naudingų funkcijų. Sujungus tuo metu egzistavusius jau tris skirtingus metodus, buvo tikimasi, kad tai padės lengviau susitvarkyti su metoduose kilusiomis sunkiai išsprendžiamomis problemomis, kurias bus įmanoma lengviau išspręsti apjungus metodus. Sistemos modeliavimas palengvina sistemos veikimo suvokimą. Vienu modeliu sunku atvaizduoti visą sistemą, todėl visada reikalingi sudėtiniai modeliai, kurie sujungti vienas su kitu norint suprasti bet kurią sistemą. Programinės įrangos sistemos dažnai reikalauja kalbos, kuri gali atvaizduoti skirtingus sistemos architektūros pjūvius jai plečiantis programinės įrangos vystymosi gyvavimo ciklo metu.

UML (angl. *Unified Modeling Language*) [1] [2]yra pramonės standartas, skirtas programinės įrangos sistemos artefaktams vizualizuoti, detalizuoti, konstruoti ir dokumentacijai kurti. UML palengvina komunikaciją ir sumažina painiavą tarp projekto suinteresuotųjų šalių. Kalbos išraiškingumas leidžia vartotojams modeliuoti viską nuo organizacijos informacinių sistemų ir paskirstytų internetu pagrįstų (angl. *web-based*) aplikacijų iki realiojo laiko sistemų. UML standartu galima aprašyti sistemos reikalavimus, apimančius tokius konceptualius dalykus, kaip verslo procesai ir sistemos funkcijos, taip pat ir tokius konkrečius dalykus, kaip klasės, parašytos specifine programavimo kalba, duomenų bazių schemos, ir perpanaudojami programinės įrangos komponentai.

Diagramos naudojamos sistemai iš skirtingų perspektyvų vizualizuoti (žr. 1.1. pav.) [3]. Sudėtinės sistemos visuma sunkiai suvokiama iš vienos perspektyvos, todėl UML naudojama norint apibrėžti visą sistemą išlaikant jos vientisumą, tačiau skirtingos diagramos vaizduoja skirtingus sistemos aspektus. Tam tikra diagrama atvaizduoja tik dalį sistemos modelio ir nebūtinai visą, todėl labai patogu, kad modelio turinys yra padalinamas į diagramas, rodančias sistemą skirtingais

pjūviais. Labai nesunku diagramose pakeisti modelį sudarančius elementus, ir taip atvaizduoti tik mažą dalį modelį sudarančios informacijos. Sistemą apibūdinantis elementų rinkinys ir sąsajos tarp jų sudaro kuriamos sistemos modelį.



1.1 pav. UML diagramų tipai, pavyzdys **[3]**

UML naudojamos 13-os rūšių diagramos [3]:

1. Klasių diagrama – struktūrinė diagrama, kuri aprašo statinę sistemos struktūrą, t.y. parodo klasių, sąsajų rinkinį ir jų ryšius. Taip pat gali būti modeliuojamos klasės, tipai, sąsajos ir ryšiai tarp jų.
2. Objektų (angl. *object*) diagrama - struktūrinė diagrama, kuri parodo objektų rinkinį, ir jų ryšius. Taip pat gali būti modeliuojami klasių objektai, apibrėžti konfigūracijų klasių diagramose, kurios yra svarbios kuriamai sistemai.
3. Komponentų (angl. *component*) diagrama - struktūrinė diagrama, kuri parodo išorinių sąsajų, įtraukiant ir jungtis, ir vidinę komponento struktūrą. Naudojamamos modeliuojant svarbius sistemos komponentus ir jų naudojamas sąsajas.
4. Kompozicinės (angl. *composite*) struktūros diagrama – tai struktūrinė diagrama, kuri parodo išorines sąsajas ir vidinę struktūrizuotos klasės sudėtį. Naudojama modeliuojant klasių ar komponentų savybes ir santykius aptariamame kontekste.
5. Panaudojimo atvejų diagrama – elgsenos diagrama, kuri aprašo aktorius ir jų santykius iš vartotojo perspektyvos. Taip pat gali būti modeliuojama sąveika tarp sistemos ir vartotojų ar kitų išorinių sistemų.
6. Sekų (angl. *sequence*) diagrama - elgsenos diagrama, kuri aprašo objektų sąveiką ir jų elgseną laike pranešimų pagalba. Taip pat gali būti modeliuojama sąveika tarp objektų, kur svarbus sąveikų eiliškumas ir tvarka.
7. Komunikacijos (angl. *communication*) diagrama – elgsenos diagrama, kuri parodo sąveiką, pabrėžiant objektų, siunčiančių ir gaunančių žinutes, struktūrą. Taip pat gali būti modeliuojami objektų sąveikavimo būdai ir tam reikalingos jungtys.
8. Būsenų (angl. *state*) diagrama – elgsenos diagrama, kuri parodo objektų būsenas ir šių pasikeitimus laiko atžvilgiu. Naudojama modeliuojant objektų gyvavimo ciklo būsenas ir veiksniai, galintys joms daryti įtaką.
9. Veiklos diagrama – elgsenos diagrama, kuri aprašo sistemos veiklų sekas tam tikrame detalumo lygyje. Taip pat gali būti modeliuojami nuosekliai ir paraleliai atliekami veiksmai kartu su sistema.
10. Diegimo (angl. *deployment*) diagrama – struktūrinė diagrama, kuri parodo santykius tarp susikirtimų taškų rinkinių, artefaktų, klasių ir komponentų. Taip pat gali būti modeliuojamas sistemos paleidimas realiojo laiko situacijoje.

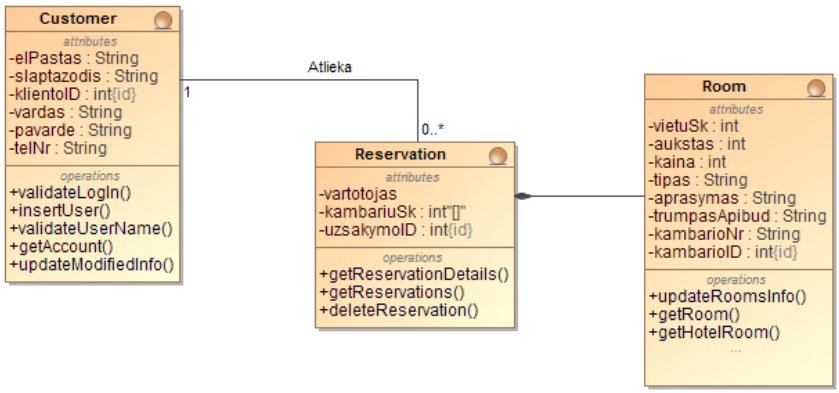
11. Paketų (angl. *package*) diagrama – struktūrinė diagrama, kuri aprašo modelio struktūrą paketais. Naudojama modeliuojant hierarchinę klasių ar komponentų grupių struktūrą.
12. Laiko (angl. *timing*) diagrama – elgsenos diagrama, kuri aprašo pranešimų sąveiką tam tikru laiko momentu. Gali būti modeliuojama sąveika tarp objektų, kuriems laikas yra labai svarbus.
13. Sąveikų apžvalgos (angl. *interaction overview*) diagrama – elgsenos diagrama, kuri apjungia veiklos ir sekų diagramų aspektus. Taip pat naudojama pavaizduoti vienoje vietoje sekų, komunikacijos ir laiko diagramas ir užfiksuojant svarbias jų tarpusavio sąsajas.

1. Klasių diagrama

Kaip objektinėse sistemose klasė yra pati svarbiausia dalis, taip ir UML diagramose svarbiausia ir dažniausiai naudojama yra klasių diagrama. Tai struktūrinė diagrama, skirta aprašyti statinę sistemos struktūrą, kurioje gali būti vaizduojamos klasės, tipai, interfeisai ir ryšiai tarp jų. Sistemos struktūra sudaryta iš dalių, vadinamų objektais, rinkinio. Klasės apibūdina skirtingus sistemoje esančius objektų tipus, o klasių diagramos parodo šias klases ir jų ryšius. Klasių diagramos apibūdina skirtingus objektų tipus, kurie yra reikalingi išpildyti sistemos panaudojimo atvejuose aprašytoms pagrindinėms sistemos funkcijoms. Didelėse sistemose, turinčiose daug susijusių klasių, klasės yra grupuojamos sukuriant klasių diagramas. Taip pat gali būti kuriamos tokios sudėtingesnės klasių formos, kaip šablonai (angl. *templates*), kurie sistemos projektavimą gali padaryti dar efektyvesnį.

Sudėtingesnės klasės gali būti apibendrintų klasių tipu arba gali turėti kitos klasės objektus, atsižvelgiant į tai, kokie stiprūs ryšiai yra tarp tų dviejų klasių. Abstrakčios klasės padeda iš dalies nusakyti klasės elgseną, t. y. leidžia kitoms specifinėms klasėms pasiimti reikalingą ir tinkančią informaciją iš abstrakčios klasės. Interfeisai padeda atsisakyti priklausomybių, kurias turi abstrakčios klasės. Jie aprašo klasės elgseną (operacijas, metodus), tačiau jų nerealizuoja. Realizacija yra atliekama konkrečioje klasėje. Taip pat klasės diagramoms galima pritaikyti apribojimus, kurie apibūdina, kaip klasės objektai gali būti naudojami su OCL (angl. *Object Constraint Language*). Naudojant klasių šablonus galima pernaudoti juose esančią bendrinę ir daugkartinio naudojimo informaciją.

Standartiškai stačiakampiu brėžiama klasių diagrama turi tris dalis: klasės vardą viršuje, atributus arba klasę, sudarančią informaciją per vidurį, ir klasės elgseną nusakančios operacijos ar metodai apačioje. Diagramą gali sudaryti nebūtinai visos trys paminėtos sekcijos, kai kurios dalys gali būti paslėptos, norint akcentuoti tik svarbiausią ir esminę klasės informaciją.



1.2 pav. Klasių diagramos pavyzdys

Kaip prieš tai minėta, klasės diagramą sudaro trys dalys. Viršuje pateiktoje diagramoje (žr. 1.2 pav.) pavaizduotos trys klasės – „Customer“, „Reservation“ ir „Room“. Visas tris klases sudaro visos trys klasę galinčios sudaryti dalys – klasės vardas, atributai ir operacijos. Atributų skiltyje pavaidzuoti duomenys, kuriais apibūdinama klasė „Reservation“: koks vartotojas rezervuoja

kambarį, kiek kambarių turi rezervuojamas kambarys ir koks yra užsakymo numeris. Taip pat prie kiekvieno atributo pavadinimo pateikiamas ir to atributo tipas, pavyzdžiui, ar tai bus datos formato duomenys, ar skaičius, ar žodinė informacija. Prie klasės operacijų skilties pateikiamos numatytos operacijos, kurios galimos atlikti su minėta klase, o taip pat ir naudojamos kitose diagramose, kai reikia panaudoti klasės atliekamas funkcijas. Taip klasės diagramos informacija gali būti panaudota kitose diagramose. Ir norint užtikrinti šios keliose diagramose naudojamos informacijos tikslumą, reikia tarpusavyje suderinti diagramas. Tik suderintose skirtingose kuriamos sistemos diagramose gali būti pateikiama tiksli ir aiški informacija, skirtingų tipų diagramose pateikiama informacija nesidubliuoja ir atitinka sistemos funkcijas.

2. Objektų (angl. *object*) diagrama

Objektai yra pagrindinė objektinės sistemos sudedamoji dalis. Objektų diagrama – tai struktūrinė UML diagrama, skirta modeliuoti objektų rinkinius ir jų ryšius. Dažniausiai modeliuojami objektai yra klasių diagramose. Sistemos funkcionavimo metu, sistemoje objektų pagalba veikia suprojektuotos klasės. Palyginus su klasių diagramomis, objektų diagramose žymėjimo sistema yra labai paprasta. Objektų diagramos yra ypatingai naudingos, kai norima apibūdinti, kaip objektai sąveikaus su sistema pagal tam tikrą scenarijų, neatsižvelgiant į tai, kad yra pakankamai ribotas diagramos objektų žymenų skaičius. Objektų diagrama padeda sudaryti loginį modelio vaizdą, taip papildant sukurtas klasių diagramas.

3. Komponentų (angl. *component*) diagrama

Projektuojant programinės įrangos sistemą nepradedama jos kurti tiesiog pagal reikalavimus iškart apibrėžiant, kokios bus klasės. Pirmiausia suplanuojamos stambios sistemos dalys, kurios sukuria sistemos pradinę architektūrą, ir valdomos tų dalių priklausomybės, kompleksiškumas. Komponentų diagrama – tai struktūrinė diagrama, skirta modeliuoti sistemos išorines sąsajas ir vidinę komponento struktūrą. Komponentų diagrama ir joje vaizduojami komponentai naudojami susisteminti programinės įrangos dalis, kurios bus valdomos, pernaudojamos ar apkeičiamos.

UML komponentų diagramos modeliuoja sistemos komponentus ir taip formuoja modelio diegimo vaizdą. Diegimo vaizdas apibūdina, kaip sistemos dalys yra susistemintos į modulius ar komponentus, ir nesunkiai padeda susidoroti valdant sistemos architektūros sluoksnius.

Komponentas yra atskira, pernaudojama ir pakeičiama programinės įrangos dalis. Komponentus galima traktuoti kaip atskirus kūrimo blokus, kuriuos galima tarpusavyje jungti norint sukurti didesnius sistemos komponentus, taip formuojant ir kuriant programinę įrangą. Dėl tokių sąvybių, komponentai gali būti skirtingų dydžių nuo mažiausio dydžio kaip klasė, iki didžiausių, tokių kaip subsistemos.

Geriausi pavyzdžiai, kokie gali būti komponentai, yra elementai, kurie atlieka pagrindinį funkcionalumą ir gali būti dažnai naudojami įvairiose sistemos dalyse. UML kalboje komponentai gali turėti tas pačias funkcijas kaip ir klasės, t. y. jungtis arba būti susiję su kitomis klasėmis ar komponentais, realizuoti (angl. *implement*) sąsajas, turėti operacijas ir t.t. Taip pat veikdami kaip sudėtinės struktūros, gali turėti jungtis ir atvaizduoti vidinę struktūrą. Lyginant komponentus su klasėmis, didžiausias skirtumas yra tai, kad komponentus sudaro didesnės atsakomybių sritys nei klasėse.

Komponentai, kaip pagrindinės sistemos sudedamosios dalys sistemos projektavime, privalo turėti sąlyginai nestiprius ryšius (angl. *louse coupling*) su kitomis sistemos dalimis tam, kad pakeistas komponentas neturėtų didelės įtakos visam likusiam sistemos funkcionalumui.

4. Kompozicinės (angl. *composite*) struktūros diagrama

Kartais klasių ar sekų diagramos neparodo visų svarbių sistemos bruožų. Kompozicinės struktūros diagramos padeda užpildyti šias trūkstamas vietas. Tai struktūrinės diagramos, kurios parodo išorines sąsajas ir vidinę struktūrizuotos klasės sudėtį. Šios diagramos parodo, kaip objektai

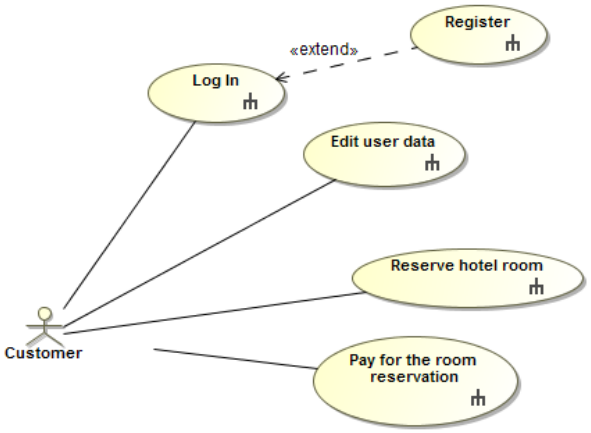
sukuria globalų sistemos paveikslą. Diagramos naudojamos modeliuojant klasių ar komponentų savybes, diagramos parodo, kaip klasės objektai tarpusavyje sąveikauja, arba kaip objektai atlieka savo užduotį diagramoje. Kompozicinės struktūros diagramos yra pakankamai sudėtingos ir išplėstos, bet jos puikiai tinka modeliuoti tokioms specifinėms situacijoms, kaip vidinės struktūros, jungtys, bendradarbiavimas (angl. *collaborations*). Kompozicinės struktūros parodo sistemos dalių vaizdą ir formuoja loginį sistemos modelio vaizdą.

Vaizduojant klasės vidinę struktūrą, dėmesys sutelkiamas į klasės turinį, o vaizduojant klasės jungtis dėmesys sutelkiamas į tai, kaip klasė naudojama kitų klasių. Jungtis yra sąveikos vieta, kurioje klasė jungiasi prie išorinių sistemos dalių.

5. Panaudojimo atvejų diagrama

Tai dažniausiai naudojamas ir žinomas UML diagramų tipas. Tai elgsenos diagramos, kurios aprašo aktorius ir jų santykius iš vartotojo perspektyvos. Taip pat gali būti modeliuojama sąveika tarp sistemos ir vartotojų ar kitų išorinių sistemų. Šios diagramos dažniausiai naudojamos kaip pačios pirmosios modeliuojant sistemą, nes jose vaizduojama tokia svarbiausia informacija, kaip vartotojo reikalavimai ar pagrindinės sistemos funkcijos, kurios yra pačios svarbiausios visame modelyje ir jų pagrindu kuriamos kitos sistemos funkcionalumo ar dizaino diagramos. Šios diagramos apibrėžia sistemos funkcionalumą, t. y. ką sistema turi daryti, tačiau nepasako, ko sistema neatlieka, ir tik po to kuriamos kitos diagramos, apibrėžiančius visus likusius reikalavimus. Panaudojimo atvejų diagrama puikiai aprašo tokius pagrindinius objektinės sistemos aspektus, kaip sistemos vystymas, dizainas, testavimas ar dokumentacija. Šios diagramos puikiai apibūdina sistemos reikalavimus iš išorės, t. y. iš vartotojo perspektyvos, pateikia grafinį sistemos aktorių, skirtingų funkcijų vaizdą, ir kaip tai sąveikauja tarpusavyje.

1.3 paveiksle pateikta panaudojimo atvejų diagrama, kuri vaizduoja, ką vartotojas gali atlikti kuriamoje informacinėje sistemoje. Šioje diagramoje vartotoją atitinka žmogeliuko figūrėlė, o iš jos išeinančios linijos link ovalų simbolizuoja, ką vartotojas sistemoje gali atlikti, kokios jo funkcijos naudojantis informacine sistema. Ovaluose aprašomos funkcijos, kurias gali atlikti diagramoje vaizduojami veikėjai, šiuo atveju vartotojas. Ši panaudojimo atvejų diagrama su kitomis UML naudojamomis informacinei sistemai kurti diagramomis siejasi tuo, kad joje naudojami elementai, tokie kaip aktoriai, panaudojimo atvejai, taip pat gali būti naudojami kitose diagramose, pavyzdžiui aktorius perpanaudojamas sekų diagramoje, todėl reikia užtikrinti, kad jei tas pats elementas pasikartoja skirtingose diagramose, nors jis turi tą pačią prasmę ir reikšmę modeliuojamoje sistemoje, tai jis turi būti atvaizduotas analogiškai, o diagramos tarpusavyje turi būti suderintos.



1.3 pav. Panaudojimo atvejų diagramos pavyzdys

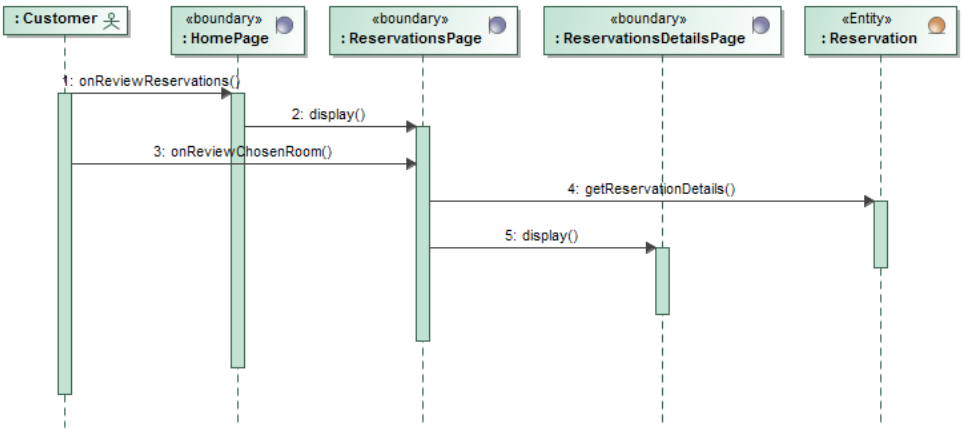
6. Sekų (angl. *sequence*) diagrama

Panaudojimo atvejai aprašo, ką sistema turi daryti, o klasių pagalba apibūdinamos skirtingos sistemos struktūros dalys. Trūkstama dalis, kurios reikia, kad sistema pilnai funkcionuotų, yra sekų diagramos, kurių pagalba nusakoma, kaip sistema iš tikrųjų atliks numatytas funkcijas. Taigi sekų diagrama yra elgsenos diagrama, kuri aprašo objektų sąveiką ir jų elgseną laike pranešimų pagalba, o taip pat šiomis diagramomis gali būti modeliuojama sąveika tarp objektų, kur svarbus sąveikų eiliškumas ir tvarka. Sekų diagramos yra labai svarbios sąveikų diagramų grupėje, kuri nusako, kaip sistemos dalys sąveikauja kartu ir formuoja bendrą loginį sistemos vaizdą.

Dažniausiai sekų diagramos nusako sistemos dalių vykdomų veiksmų eigą ir eiliškumą. Naudojant sekų diagramas galima nusakyti, kurie veiksmai bus suaktyvinti įvykdžius tam tikrus panaudojimo atvejus, ir kokiu eiliškumu jie bus įvykdyti. Sekų diagramos gali būti sudarytos iš daugybės sąveikų veiksmų, bet šių diagramų esmė yra kaip paprastai ir efektyviai jose galima pateikti sąveikų įvykių eiliškumą.

Sekų diagrama sudaroma iš sistemos dalių rinkinio, kurio dalys sąveikauja viena su kita tam tikroje sekoje. Labai svarbu, kurioje sekų diagramos vietoje yra atvaizduota tam tikra sistemos dalis. Atvaizdavimas vertikaliai neturi lemiamos įtakos, bet labai svarbu, kad sistemos dalys būtų tvarkingai išdėstytos horizontaliai ir dalys nepersidengtų viena su kita. Kiekviena vaizduojama dalis turi atitinkamą gyvavimo liniją (angl. *lifeline*), einančią iki diagramos apačios. Sistemos dalies gyvavimo linija pažymi būseną, kuri žymi, kad tam tikra sistemos dalis egzistuoja tam tikrame sekos aprašomame taške ir yra svarbi kol yra kuriama arba ištrinama vykdant seką.

1.4 paveiksle pavaizduoje sekų diagramoje matoma, kaip tarp vartotojo ir rezervacijos objektų vyksta sąveika. Rodyklės vaizduoja kokie pranešimai perduodami, ir tarp kokių objektų. Vykstančių veiksmų seka atitinkamai pagal eiliškumą sužymėta skaičiais virš rodyklių. Yra kelių tipų rodyklės: juoda rodyklė su pilnaviduriu galu atitinka sinchroninę žinutę. Šioje diagramoje tokia rodyklė vaizduoja vartotojo perduodamą žinutę rezervacijos objektui. Kita rodyklė – tai grįžtamojo pranešimo rodyklė, kuri atitinka rezervacijos objekto grąžinamą pranešimą vartotojo objektui kaip atsaką. Taip vaizduojamas apsikeitimas pranešimais tarp objektų kuriamoje sistemoje. Taip pat šioje diagramoje pavaizduoti objektai gali būti ir kitose kuriamos sistemos modelio diagramose, taigi šioje vietoje taip pat reikalingas naudojamų elementų suderinamumo užtikrinimas kaip ir prieš minėtose keliose diagramose.



1.4 pav. Sekų diagramos pavyzdys

7. Komunikacijos (angl. *communication*) diagrama

Komunikacijos diagrama aprašo labai panašią informaciją kaip ir sekų diagrama. Sekų diagramų pagrindinis tikslas yra parodyti įvykių tvarką ir eiliškumą tarp sistemos dalių, įtrauktų į tam tikrą procesą. Komunikacijos diagramos leidžia pažvelgti į sistemos dalių sąveikavimą iš kitos

perpektyvos, kada dėmesys sutelkiamas į ryšius tarp šių dalių. Tai elgsenos diagrama, kuri parodo sąveiką, pabrėžiant siunčiančių ir gaunančių žinutes objektų struktūrą. Taip pat gali būti modeliuojami objektų sąveikavimo būdai ir tam reikalingos jungtys. Komunikacijos diagramos labai gerai aprašo, kokie ryšiai reikalingi tarp sistemos dalių norint perduoti pranešimą. Sekų diagramoje ryšiai tarp sistemos dalių yra suprantami kaip faktas, kai pranešimas jau perduotas. Komunikacijos diagramos intuityviai numato, kad tarp sistemos dalių yra reikalingas ryšys pranešimams perduoti. Komunikacijos diagramoje į sąveiką įtrauktų įvykių eiliškumas yra antraeilė informacija.

Komunikacijos diagramą sudaro trys dalykai: sistemos dalys, komunikacijos ryšiai tarp jų, ir pranešimai, perduodami per tuos ryšius. Sistemos dalys žymimos stačiakampiu, kuriame įrašytas klasės pavadinimas ir jos informacija. Komunikacijos ryšys sujungia sistemos dalis, tarp kurių siunčiami pranešimai.

Sekų ir komunikacijos diagramos yra itin panašios, todėl UML įrankiai gali automatiškai pakeisti diagramos tipą iš vienos į kitą. Didžiausias skirtumas tarp šių diagramų yra asmeninė vartotojo nuomonė ir kuri diagrama jam atrodo priimtinesnė.

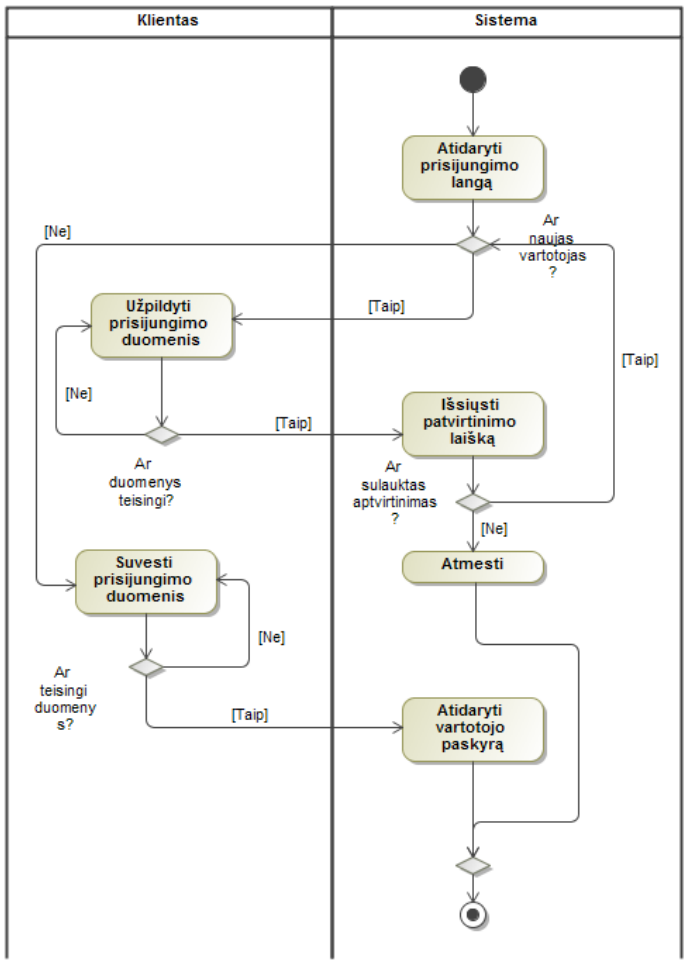
8. Būsenų (angl. *state*) diagrama

Veiklos ir sąveikos diagramos yra naudingos apibūdinant elgseną, tačiau dažnai dar reikalinga ir sistemos ar objekto būsena kaip svarbus elgsenos faktorius. Tuomet naudojamos būsenos diagramos, kada reikia modeliuoti objekto gyvavimo ciklo būsenas ir įvykius, sukeliančius būsenos pasikeitimą, ar joms galinčius daryti įtaką veiksnius. Būsenų diagrama yra elgsenos diagrama, kuri parodo objektų būsenas ir šių pasikeitimus laiko atžvilgiu.

9. Veiklos diagrama

Prieš tai minėtose panaudojimo atvejų diagramose vaizduojama, ką sistema turi atlikti, o šiose veiklos diagramose apibrėžiama, kaip sistema turi tai padaryti. Veiklos diagrama yra elgsenos diagrama, kuri aprašo sistemos veiklų sekas tam tikrame detalumo lygyje. Taip pat gali būti modeliuojami nuosekliai ir paraleliai atliekami veiksmai kartu su sistema. Veiklos diagramose grafiškai vaizduojami sistemoje vykstantys darbo eigos srautai. Diagramos itin tinkamos aprašyti verslo procesus. Tai yra vienintelis diagramos tipas, kuriuo galima aprašyti sistemoje vykstančius procesus. Veiklos diagramas nesudėtinga tiek kurti, tiek skaityti, nes jos labai panašios į plačiai žinomas ir naudojamas struktūrines schemas, todėl aprašomi procesai nesunkiai suprantami plačiai auditorijai. Veiklos diagramos dar gali būti naudojamos kaip alternatyvos mechanizmų būsenų diagramoms.

1.5 pav. vaizduojama veiklos diagrama, kurioje pateikiama veiksmų seka, kokius veiksmus reikia atlikti sistemos vartotojui norint prisijungti prie sistemos. Diagrama pradedama kurti diagramos objektu, žyminčiu proceso pradžią, tuomet aprašomi veiksmai, kurie reikalingi aprašyti norimai veiklai sistemoje, visi diagramos objektai sujungiami. Veiksmų sekos pabaigą žymi atitinkamas pabaigą reiškiantis diagramose naudojamas objektas. Kaip ir prieš tai aprašytose keliose diagramose, šioje diagramoje naudojami objektai taip pat turi būti suderinti su kitose diagramose analogiškai naudojamais tokiais pačiais elementais, kurie reiškia tą patį objektą ar procesą visoje kuriamoje sistemoje, tik pavaizduoti skirtingose diagramose iš skirtingų aspektų. Tokių pasikartojančių elementų užtikrinimas, kad skirtingose diagramose naudojami tie patys elementai, identifikuojantys tą pačią sistemos vietą, itin reikalingas, kad sistema būtų korektiškai modeliuojama.



1.5 pav. Veiklos diagramos pavyzdys

10. Diegimo (angl. *deployment*) diagrama

Visos UML diagramos aprašo sistemą iš įvairių požiūrių, tačiau nei viena neaprašo fizinio sistemos pjūvio. Fizinis vaizdas siejamas su fiziniais sistemos elementais, tokiais kaip vykdomieji programinės įrangos failai ir techininė įranga, kurioje jie veikia. Taigi diegimo diagrama pateikia fizinį sistemos vaizdą, taip apibūdinant programinę įrangą realiame pasaulyje, parodant kaip šiai priskiriamos techninei įrangai ir kaip tai perduodama. Tai struktūrinė diagrama, kuri parodo santykius tarp susikirtimų taškų rinkinių, artefaktų, klasių ir komponentų. Taip pat gali būti modeliuojamas sistemos paleidimas realiojo laiko situacijoje. Diegimo diagrama turi būti sudaryta iš sistemos dalių, kurios svarbios auditorijai. Jeigu svarbu parodyti techninę įrangą, integruotą programinėje įrangoje, operacinę sistemą, programos vykdymo aplinką (angl. *runtime environment*), ar net sistemos įrenginių tvarkykles (angl. *device drivers*), tai šie komponentai ir turi būti pavaizduoti kuriamoje diegimo diagramoje.

11. Paketų (angl. *package*) diagrama

Kada kuriama sistema vis sudėtingėja, ją pradeda sudaryti šimtai klasių. Vienas iš būdų viską struktūrizuoti yra klases suskirstyti į logiškai susijusias grupes, kurias sumodeliuoti galima UML paketų diagramomis. Tai struktūrinės diagramos, kurios aprašo modelio struktūrą paketais. Šio tipo diagramos naudojamos modeliuojant hierarchinę klasių ar komponentų grupių struktūrą. Dažniausiai paketų diagramos naudojamos norint nustatyti paketų priklausomybes ir užtikrinti, kad pakeitus ar pašalinus vieną paketą, išliks programinės įrangos stabilumas.

Paketų diagramos naudojamos norint susisteminti bet kuriuos UML elementus, bet dažniausiai naudojama panaudojimo atvejams ar klasės elementams. Nors UML įrankis greičiausiai neturi diagramos pavadinimu „Paketų diagrama“, tačiau paketai grupuoja struktūras, kurios naudojamos sisteminti bet kuriuos UML elementus, ir kaip prieš tai buvo minėta, dažniausiai šios diagramos naudojamos sisteminti klases klasių diagramose.

12. Laiko (angl. *timing*) diagrama

Kalbant apie sekų ar komunikacijos diagramas, atitinkamai vaizduojama arba pranešimų tvarka, arba ryšiai tarp sistemos dalių, tačiau nei vienoje iš šių diagramų nevaizduojama detali laiko informacija. Sąveikų laikas dažniausiai siejamas su realaus laiko arba integruotomis sistemomis, bet nebūtinai apibrėžiamas tik šiomis sritimis. Laiko diagrama yra elgsenos diagrama, kuri aprašo pranešimų sąveiką tam tikru laiko momentu. Taip pat gali būti modeliuojama sąveika tarp objektų, kuriems labai svarbus yra laikas. Poreikis užfiksuoti tikslią sąveikos laiko informaciją gali būti svarbus neatsižvelgiant į tai, kokia sistema kuriama.

Laiko diagramoje kiekvienas įvykis turi laiko informaciją, susijusią su jo pradžia, kiek laiko užtrunka kitai sistemos daliai gauti pranešimą, ir kiek ilgai gaunančioji pranešimą dalis turi jo laukti ir nekeisti būsenos.

Laiko diagramos panašios į elektroninį prietaisą (angl. *logic analyzer*), kuris atvaizduoja pasikartojančius signalus iš skaitmeninės sistemos arba skaitmeninės grandinės. Jis fiksuoja įvykių seką, kada jie įvyksta elektros grandinėje (angl. *electronic circuit board*). Prietaiso rodmenys parodo laiką, kada skirtingos grandinės dalys yra tam tikrose būsenose, ir elektroninius signalus, kurie sužadins šioms būsenoms pokyčius.

13. Sąveikų apžvalgos (angl. *interaction overview*) diagrama

Sąveikų apžvalgos diagramos skirtos pateikti visą sistemos sąveikų vaizdą. Tai elgsenos diagramos, kurios apjungia veiklos ir sekų diagramų aspektus. Sąveikų apžvalgos diagramos leidžia peržiūrėti sąveikų bendrą veiklą įgyvendinant sistemos reikalavimus, pavyzdžiui, pavaizduotus panaudojimo atvejuose. Šios diagramos naudojamos pavaizduoti vienoje vietoje sekų, komunikacijos ir laiko diagramas ir užfiksuojant svarbias jų tarpusavio sąsajas. Sekų, komunikacijos ir laiko diagramos sutelkia dėmesį į specifines detales, susijusias su sąveikas sukuriančiais pranešimais, o sąveikų apžvalgos diagramos apjungia visas šias sąveikas į vieną didelį sąveikų paveikslą, kuris vaizduoja tam tikrą sistemos išpildytą reikalavimą.

Sąveikų apžvalgos diagrama labai panaši į veiklos diagramą, išskyrus tai, kad kiekvienas veiksmas yra sąveika su savomis teisėmis. Ši diagrama apjungia kartu skirtingas sistemos sąveikas vienoje diagramoje, kuri sukuria didžiausią prasmę aprašomoms sąveikoms, taip pavaizduojant sąveikavimą kuriant ir išpildant sistemos reikalavimus.

Sąveikų apžvalgos diagrama santykinai apjungia kartu sekų, komunikacijos ir laiko diagramas, taip parodant trijų diagramų bendrą sąveikos vaizdą globalesniu požiūriu visoje sistemoje.

Modelio elemento naudojimas keliose diagramose

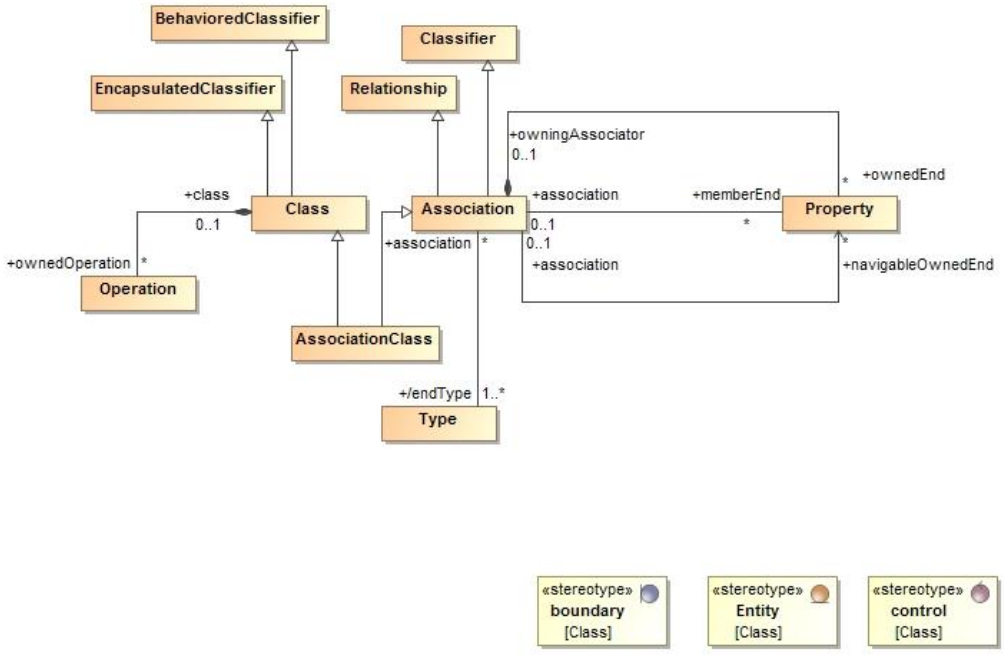
Iš prieš tai aprašytų diagramų, dažniausiai naudojamos klasių, panaudojimo atvejų, sekų, veiklos. Šiose diagramose elementai gali būti perpanaudojami iš jau kitose diagramose panaudotų elementų. Pavyzdžiui, klasių diagramoje panaudotos klasės gali būti panaudojamos kaip gyvavimo linijos sekų diagramose, kai modeliuojamas detalesnis sistemos veikimas ir vykstantys procesai. Taip pat dažnai panaudojimo atvejams kuriamos įvairios diagramos, kurios nusako to panaudojimo atvejo detalesnį veikimą. Dažnai panaudojimo atvejam kuriamos veiklos diagramos, sekų diagramos. Kai taip modeliuojamos sistemos ir diagramos tampa stipriai priklausomos viena nuo kitos, šių diagramų elementai perpanaudojami keliose diagramose – modeliuojama viena sistema ir tam tikras elementas turi tikslią reikšmę sistemoje, o skirtingose diagramose elementas vaizduojamas iš skirtingų požiūrio

taškų. Taigi dažnai skirtingose diagramose pradedami naudoti skirtingi elementai, nors jų reikšmė visoje modeliuojamoje sistemoje yra tokia pati. Taip atsiranda neatitikimų modelyje, tiek diagramos, tiek modelis tampa netikslūs, nekorektiški. Norint užtikrinti diagramų elementų suderinamumą, reiktų taikyti taisykles, tikrinančias modelio taisyklingumą. Programose tokių taisyklių nėra arba yra mažai, o ir pačios taisyklės tikrina įprastas taisykles, tokias kaip ar elementas turi pavadinimą. Tokių taisyklių yra per mažai ir norint užtikrinti geresnį modelio elementų suderinamumą, reikia taikyti sudėtingesnes taisykles, kai tikrinamos diagramos ir tų diagramų naudojami elementai.

1.2.2. UML metamodeliai

UML kalba remiasi metamodeliais, t. y. kiekvienas UML naudojamas elementas turi savo metaklasę, kuri turi atskirus nustatymus. Naudojant metaklases gali būti aprašomos sąlygos, apribojimai, skirti UML braižytiems modeliams tikrinti. Toliau darbe pateikiami dažniausiai naudojamų UML diagramų metamodeliai.

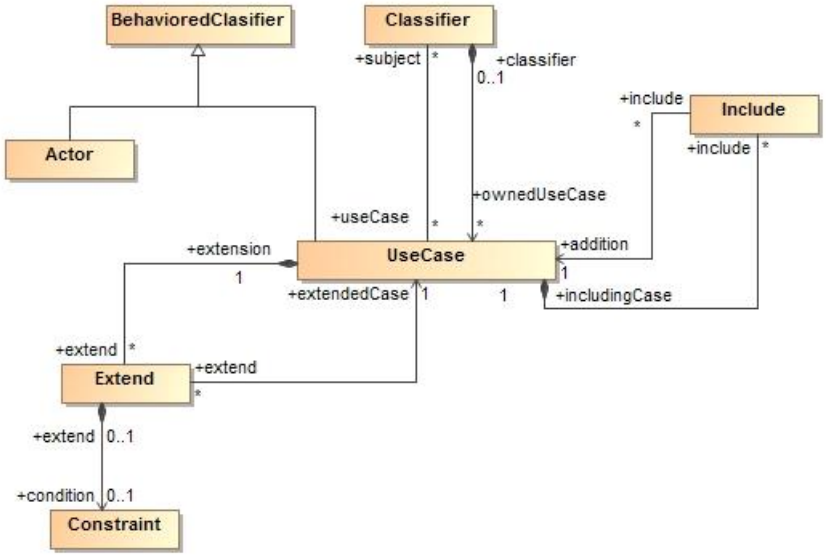
Paveiksle 1.6 pateiktas metaklasės *Class* ir su ja susijusių kitų metaklasių modelis. Metaklasė *Class* yra konkreti *EncapsulatedClassifier* ir *BehavioredClassifier* realizacija. Šios metaklasės tikslas yra specifikuoti objektų klasifikaciją ir požymius, kurie charakterizuoja tų objektų struktūrą ir elgseną. Šios metaklasės požymiai yra *Properties*, *Operations*, *Receptions*, *Ports* ir *Connectors*. Šiame 1.6 pavyzdyje pateikti ne visi požymiai. Pateikti tik tie, kurie bus naudojami šiame darbe. Taip pat šiame paveiksle pateikta, kaip su metaklase *Class* sąveikauja klasių modelyje naudojami ryšiai. Viena tokių metaklasių yra *Association*. Ši metaklasė apibrėžia, kaip pateikiami ryšiai tarp jau aprašytų egzempliorių. *AssociatedClass* tuo pačiu apibrėžia ir *Association*, ir *Class* kaip vieną visumą. *Association* apibrėžia semantinį santykį, kuris gali atsirasti tarp egzempliorių. Ši metaklasė turi mažiausiai du galus, kuriuos nusako *Properties* metaklasė, kada abu galai turi tam tikrą tipą, kuris matomas modeliuojamame modelyje pasirinkus tam tikrą tipą. Taip pat šiame darbe naudojamos išbaigtumo diagramose naudojamos *boundary*, *entity*, *control* klasės. Jos aprašomos kaip tam tikras stereotipas, tačiau jos vistiek laikomos metaklasės *Class* dalimi.



1.6 pav. Klasių metamodelis

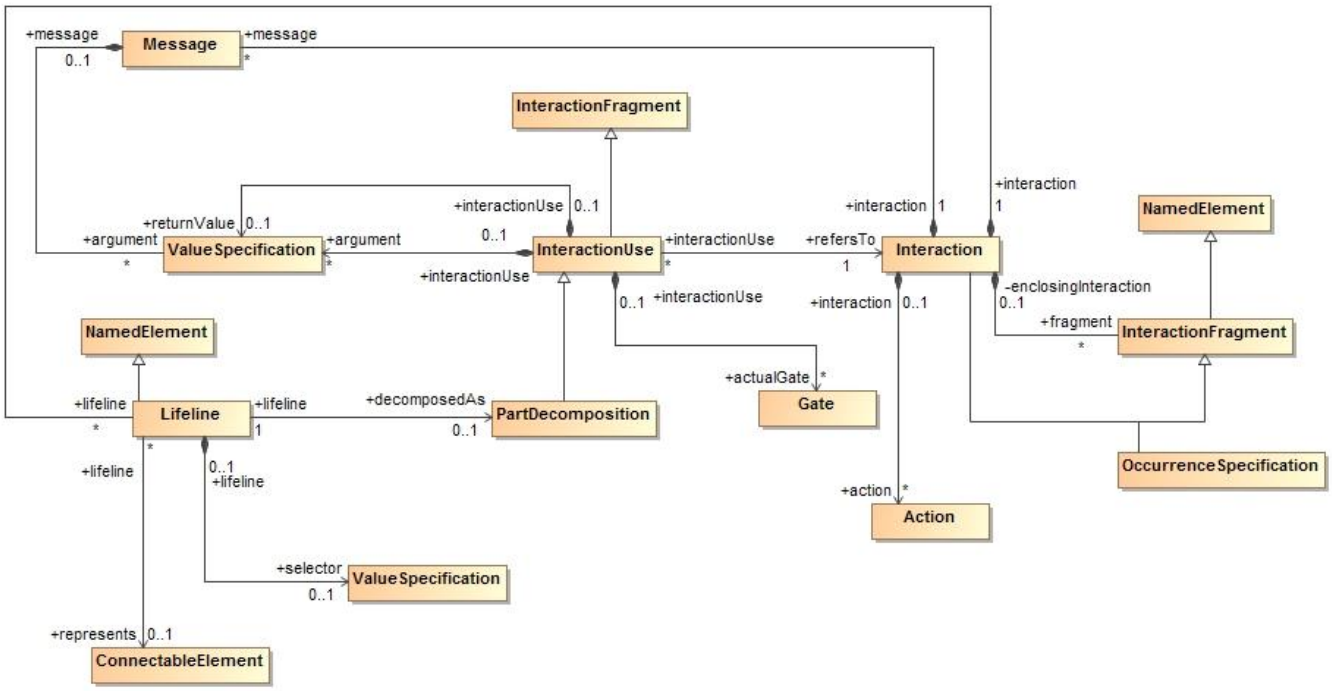
Panaudojimo atvejai yra priemonė sistemos reikalavimams surinkti, t. y. aprašyti, ką sistema turėtų atlikti iš modeliuojamos sistemos vartotojo požiūrio taško. Pagrindinės metaklasės šiame 1.7 paveiksle pateiktame metamodelyje yra *Actors*, *UseCases* ir kt. Kiekvienas metaklasės *UseCase*

subjektas pristato sistemą nurodant, kuriam iš *UseCase* yra taikomas. Vartotojai ir kitos sistemos, kurios gali sąveikauti su subjektu, pristatomi kaip *Actors*. *UseCase* yra elgsenos specifikacija. *UseCase* egzempliorius priskiriamas naujai besiformuojančio įvykio elgsenai, kuri atitinka atitinkamą *UseCase* metaklasę. Tokie egzemplioriai apibūdinami kaip *Interactions*. *UseCase* galima taikyti bet kokiam subjektų skaičiui. Kada *UseCase* taikomas subjektui, tuomet specifikuojamas elgsenų rinkinys, atliekamas to subjekto, kuris sukuria pastebimą rezultatą, kuris vertingas *Actors* metaklasei ir kitoms subjekto suinteresuotosioms šalims.



1.7 pav. Panaudojimo atvejų metamodelis

Interactions naudojami daugybei skirtingų situacijų. *Interactions* specifikuoja abstrakčią sintaksę, semantiką ir notaciją tokiom metaklasėm, kaip *Interaction, Interactionfragment, OccurenceSpecification, ExecutionSpecification, StateInvariant*. *Interactions* yra *Classifier* elgsenos vienetai. *Interactions* didžiausią dėmesį skiria praeinančiai informacijai, kada naudojami pranešimai *Messages* tarp *Classifier* metaklasės *ConnectableElements* metaklasių. *Lifelines* specifikuoja abstrakčią sintaksę, semantiką ir notaciją tokiom metaklasėm, kaip *Lifeline*. Sąveikos diagramoje *Lifeline* apibūdina proceso laiko liniją, kai laikas didėja puslapiu žemyn. Atstumas tarp dviejų įvykių, esančių laiko linijoje, nenusako jokio tikslaus laiko matavimo, parodomas tik faktas, kad praėjo nenulinis laiko tarpas. *Messages* specifikuoja abstrakčią sintaksę, semantiką ir notaciją tokiom metaklasėm, kaip *Message, MessageEnd, MessageOccurenceSpecification, MessageSort, Gate* ir kitom. *Message* semantika paprasta – įvykių <*sendEvent, receiveEvent*> fiksavimas. Dažniausiai sutinkamas *Interaction* diagramų tipas – sekų diagramos, kurios daugiausia dėmesio skiria *Message* apsikeitimui tarp tam tikro skaičiaus *Lifeline* metaklasių. Sekų diagramų nusakomi *Interactions* formuoja *Interactions* paketo metaklasių svarbiausią semantikos supratimą.



1.8 pav. Sekų metamodelis

1.2.3. ICONIX metodas

Teoriškai, kiekvienas UML aspektas yra potencialiai naudingas, bet praktiškai, dažnai darbuotojams atrodo, kad niekada nebūna pakankamai laiko modelio kūrimui, analizei ir projektavimui. Visada jaučiamas spaudimas kuo greičiau pereiti prie kodo kūrimo, kuo anksčiau pradėti programuoti, nes programinės įrangos projektų progresas paprastai įvertinamas suprogramuotų eilučių skaičiumi. ICONIX procesas yra minimalistinis, racionalaus požiūrio, kuris sutelkia didžiausią dėmesį į vietą, kuri yra tarp panaudojimo atvejų ir kodo. Koncentruojamasi į tai, kas turi įvykti tam tikrame gyvavimo ciklo (angl. *life cycle*) taške, kuriame pradedama dirbti: pradedama keliais panaudojimo atvejais, kuriems reikia atlikti analizę ir projektavimą.

Pagrindinės ICONIX metodo savybės [4] :

- Procesas naudoja pagrindinius UML poaibius;
- Procesas pagrindžia visą kelią iki kodo;
- Procesas atsekamas tarp etapų;
- Numato ir gerus (angl. *sunny-day*), ir blogus (angl. *rainy-day*) sistemos veikimo scenarijus;
- Daroma prielaida, kad pateikti reikalavimai yra neaiškūs, dviprasmiški, neišsamūs, netikslūs;
- Remiasi OO projektavimu iš panaudojimo atvejų;
- Puikiai veikia taikant *Agile* principus;
- Nenaudoja ilgų žyminčiųjų žodžių (angl. *buzzwords*);
- Dirbama realiai, kada panaudojimo atvejai atvaizduoja, ką vartotojai atlieka sistemoje, taip pat nėra didelių panaudojimo atvejų šablonų;
- Naudojamas praktinis požiūris, kuris buvo įrodytas dirbant su šimtais projektų.

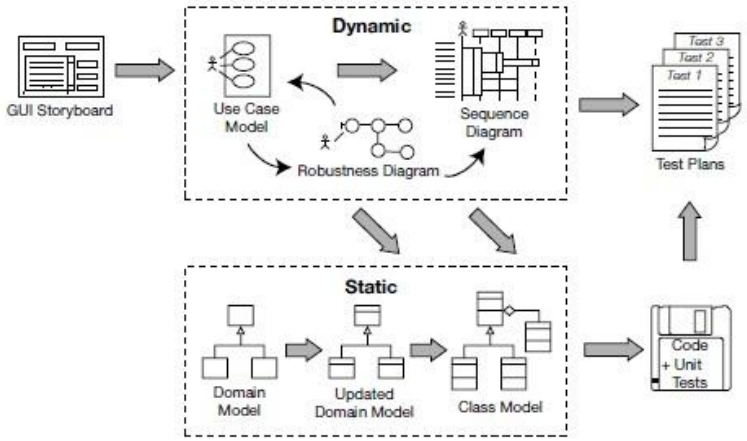
ICONIX procesas (žr. 1.9 pav.) [4] yra padalintas į dinamines ir statines darbo eigas, kurios yra pasikartojančios: galima pereiti per vieną viso proceso iteraciją iš mažos serijos panaudojimo atvejų, per visą pradinį kodą ir vienetų testus. Būtent dėl šio bruožo, ICONIX procesas labai gerai tinka Ag*ile* projektams, kuriuose reikalingas greitas grįžtamasis ryšys dėl tokių faktorių kaip reikalavimai, projektavimas, vertinimas.

ICONIX metode naudojama PĮ kūrimo eiga:

1. Reikalavimai.
    a. Funkciniai reikalavimai: apibrėžiama, ką sistema turėtų būti pajėgi atlikti. Priklausomai nuo projekto organizavimo, funkciniai reikalavimai kuriami komandos arba reiklavimai pateikiami kliento ar verslo analitikų komandos.
    b. Dalykinės srities (angl. *domain*) modeliavimas: probleminės srities supratimas nedviprasmiškais terminais.
    c. Elgsenos (angl. *behavioral*) reikalavimai: apibrėžiama, kaip vartotojas ir sistema sąveikaus. Rekomenduojama pradėti nuo vartotojo sąsajos (angl. *GUI*) prototipo ir nustatyti visus panaudojimo atvejus, kurie bus realizuoti, arba bent jau pradėti nuo panaudojimo atvejų sąrašo, kuris labai tikėtina, kad keisis, kada reikalavimai tampa labiau išanalizuoti.
    d. Gairė 1. Reikalavimų peržiūra: įsitikinama, kad panaudojimų atvejų tekstas atitinka kliento lūkesčius.
2. Preliminarus projektavimas.
    a. Išbaigtumo (angl. *robustness*) analizė: sukuriama išbaigtumo diagrama perrašant panaudojimo atvejų tekstą.
    b. Atnaujinamas dalykinės srities modelis, kol aprašomi panaudojimo atvejai ir kuriamos išbaigtumo diagramos. Čia aptinkamos trūkstamos klasės, ištaisomos dviprasmybės, pridedami atributai dalykinės srities objektams.
    c. Visoms loginėms programinės įrangos funkcijoms (valdikliams), reikalingoms panaudojimo atvejams veikti, suteikiami pavadinimai.
    d. Perrašomi juodraštiniai panaudojimo atvejai.
3. Gairė 2. Preliminaraus projektavimo peržiūra (angl. *preliminary design review*).
4. Detalizuotas projektavimas.
    a. Sekų diagramos: sudaromos sekų diagramos (kiekvienam panaudojimo atvejui po vieną sekų diagramą) detalizuotai parodant, kaip bus įvykdyti panaudojimo atvejai. Pirminė sekų diagramų funkcija yra paskirstyti funkcionavimą kuriamoms klasėms.
    b. Atnaujinamas dalykinės srities modelis, kol sudaroma sekų diagrama, pridedamos operacijos (taip pat vadinamos metodais, funkcijomis, pranešimais priklausomai nuo naudojamos programavimo kalbos) dalykinės srities objektams. Šioje stadijoje dalykinės srities objektai atitinkamai yra dalykinės srities klasės, arba esybės, taip pat dalykinės srities modelis tampa statiniu modeliu, arba klasės diagrama, tai yra lemiama dalis detaliame projektavime.
    c. Ištrinamas statinis modelis.
5. Gairė 3. Kritinė projektavimo peržiūra.
6. Vykdymas.
    a. Programavimas ir vienetų (angl. *unit*) testų rašymas: rašomas kodas ir vienetų testai, arba atvirkščiai – pirmiau rašomi vienetų testai ir tada programinis kodas.
    b. Integravimas ir scenarijaus testavimas: Pagrįsti integracijos testus kiekvienam iš panaudojimo atvejų, kad būtų ištestuotos abi - pagrindinės ir šalutinės eigos.
    c. Atliekama kodo peržiūra ir modelio atnaujinimas ruošiantis kitam vystymo darbų etapui.

Taigi šiame procese išskiriami pagrindiniai programinės įrangos kūrimo etapai pradedant nuo reikalavimų surinkimo. Kaip šio proceso autoriai teigia, tai yra minimalistinis ir racionalaus požiūrio

kūrimo procesas, kuriame aptariami pagrindiniai aspektai, reikalingi kuo greičiau pradėti praktinį programinės įrangos kūrimo etapą.



1.9 pav. ICONIX proceso grafinė iliustracija **[4]**

ICONIX ir *MagicDraw*

Šiame darbe pasirinktas taikyti *ICONIX* metodas, skirtas programinei įrangai, informacinėms sistemoms kurti. Šį metodą, naudojantį UML diagramas ir elementus, pasirinkta taikyti *MagicDraw* programoje. Taigi buvo pasidomėta, ar *ICONIX* metodą garsinanti įmonė turi kokių nors sąsajų su *MagicDraw* produktą išleidusia įmone *No Magic*.

2013 metais *No Magic* ir *ICONIX* tapo partneriais. *No Magic* yra lyderiaujanti pasaulinė įmonė, teikianti integruoto dizaino, simuliacijos, analizės paslaugas ir sprendimus, tuo tarpu *ICONIX* yra gerai žinoma įmonė, teikianti individualiai vartotojui pritaikyto proceso mokymus. Šioms įmonėms tapus partneriais, pradėti vykdyti plataus profilio mokymai, kada mokoma dirbti *MagicDraw* programa remiantis *ICONIX* programinės įrangos kūrimo principais. Buvo sukurtas unifikuotas procesas, skirtas sistemoms modeliuoti, testuoti programinę įrangą naudojant *SysML*, *UML* ir automatinių testų atvejų generavimo kombinaciją, kuri buvo pavadinta *SWISSML*. Šis procesas apima reikalavimais paremtą projekto modeliavimą įtraukiant tiek progaminę, tiek techninę įrangą, o taip pat ir automatinį visapusišką testų atvejų generavimą *MagicDraw* programoje modeliuojamoms sistemoms validuoti ir verifikuoti. Šio proceso taikymui yra specialiai sukurti mokymai, kuriuose apimamos tokios temos, kaip sistemų inžinerija (*SysML*) ir programinės įrangos inžinerija (*UML*), įtraukiant reikalavimų paskirstymą ir atsekamumą, o taip pat ir automatinių testų atvejų generavimą iš sukurto modelio naudojant *ICONIX DDT* įskiepį. Nors susijungus šioms lyderiaujančioms kompanijoms, ir apjungus jų teikiamas paslaugas buvo sukurtas naujas programinės įrangos kūrimo procesas ir pradėti vykdyti mokymai *MagicDraw* aplinkoje taikant *ICONIX* metodą, tačiau apie pačio modelio, kuriamo *MagicDraw* aplinkoje, validaciją, skirtą tikrinti *ICONIX* proceso ypatybes, nebuvo užsiminta.

Nors šioms dviems įmonėms tapus partneriais ir pradėjus bendradarbiauti buvo pristatytas jųdviejų naujas produktas ir paslaugos, tačiau *MagicDraw* vis dar nėra validacijos, skirtos tik *ICONIX* proceso kuriamo modelio korektiškumui užtikrinti.

1.2.4. OCL ir jos taikymas *MagicDraw* validacijos modulyje

OCL (angl. *Object Constraint Language*) yra formali kalba, skirta *UML* modelių išraiškoms aprašyti [5]. Paprastai šios išraiškos aprašo nekintamas sąlygas, kurios turi nekisti, kol sistema

modeliuojama, arba užklausas modelyje aprašytiems objektams. Vykdomos *OCL* užklausos visiškai neturi jokio šalutinio efekto sistemos funkcionalumui.

OCL išraiškos gali būti naudojamos aprašyti operacijas arba veiksmus, kurie juos vykdant keičia sistemos būseną. Besinaudojantys *UML* gali naudoti ir *OCL* aprašyti konkrečiai taikomus apribojimus tam tikriems konkretiems modeliams. Taip pat *UML* vartotojai gali naudoti *OCL* aprašyti *UML* modelio užklausas, kurios yra visiškai nepriklausomos nuo programavimo kalbos [6] [7].

OCL yra ne kas kita, kaip gryna specifikacijų kalba, todėl *OCL* aprašytos išraiškos visiškai neturi jokių šalutinių efektų. Kai vykdoma *OCL* aprašyta išraiška, grąžinama tik tam tikra reikšmė, niekas modelyje nekeičiama. Tai reiškia, kad sistema niekada nepakeis savo būsenos dėl *OCL* išraiškų vykdymo, net jei išraiška naudojama nusakyti būsenų pokytį (pavyzdžiui, naudojant *post-condition* išraiškas).

Kiekviena *OCL* išraiška turi tipą, todėl vadinama tipus turinti kalba (angl. *typed language*). Tiksliau, *OCL* išraiška turi atitikti kalbos taisyklių atitikties tipui. Pavyzdžiui, negalima palyginti *Integer* tipo objektų su *String* tipo objektais. Kiekvienas klasifikatorius, apibrėžtas *UML* modelyje, pateikia atskirą *OCL* tipą. Taip pat, *OCL* yra įtrauktas papildomas iš anksto nustatytų tipų rinkinys.

OCL yra tipus turinti kalba, todėl pagrindinės tipų reikšmės aprašytos tipų hierarchijoje. Ši hierarchija nustato skirtingų tipų vienas kito atžvilgiu atitikimą, t. y. negalima palyginti *Integer* tipo objekto su *Boolean* ar *String* tipo.

OCL gali būti naudojama įvairiems skirtingiems tikslams pasiekti:

- Naudojama kaip užklausų kalba;
- Aprašyti klasių nekintamumą ir klasių modelio tipams;
- Aprašyti stereotipų (angl. s*tereotypes*) tipų nekintamumą;
- Apibūdinti prieš ir po sąlygas operacijoms ir metodams (angl. *operations*, m*ethods*);
- Aprašyti tikslinius rinkinius žinutėms ir veiksmams;
- Aprašyti operacijų apribojimus;
- Aprašyti išvestas (angl. *derivation*) taisykles *UML* modelyje bet kokios išraiškos atributams.

OCL išraiška, kurioje visi tipai atitinka, vadinama galiojančia, arba pagrįsta, išraiška. O *OCL* išraiška, kada tipai neatitinka, vadinama negaliojančia, nepagrįsta išraiška. Tipų atitikimų taisyklės yra paprastos – kiekvienas tipas atitinka kiekvienam savo supertipui (angl. *supertype*); tipų atitikimas yra pereinantis: jeigu tipas1 atitinka tipas2, o tipas2 atitinka tipas3, tuomet tipas1 atitinka tipas3.

OCL išraiškos gali kreiptis į klasifikatorius, pvz. tipus, klases, sąsajas, asociacijas (kaip tipus), ir duomenų tipus. Taip pat gali būti naudojami visi atributai, asociacijų pabaigos, metodai, operacijos be šalutinio poveikio, kurie yra apibrėžti šiuose tipuose ir pan.

Validacijos taisyklė, aprašoma *OCL2.0*, dažniausiai kuriama norint aprašyti modelio elementus. Taip pat jeigu reikalaujama sprendimo rastoms klaidoms atliekant validaciją. *OCL2.0* naudojama jeigu modelio tikrinimui reikia naudoti aktyvią validaciją (angl. *active valiation*).

OCL2.0 validacijos taisyklė aprašo validacijos logiką naudojant *OCL2.0*. *OCL2.0* validacijos taisyklė gali būti naudojama aktyvios validacijos aplanke. Šiuo atveju, validacijos taisyklė bus vykdoma po bet kokių apribotų elementų, kurie yra validacijos srityje, pokyčių. Validacijos taisyklė gali būti aktyvuota net jei įvyko ir neitin svarbūs pokyčiai, todėl tai gali sulėtinti aktyvios validacijos greitį. Norint išvengti tokio sulėtinimo, galima kurti dvejetainę (angl. *binary*) validacijos taisyklę, kuri naudoja *OCL2.0* išraišką, tikrinti modelio elementus, tačiau taip pat suteikia papildomos informacijos modeliavimo įrankiui, kuris leidžia optimizuoti modelio elementų pokyčius, kurie aktyvuoja validacijos taisyklę. Pavyzdžiui, aprašant taisyklę, kuri tikrina, ar klasės turi pavadinimus, aprašius tik *OCL2.0*, bus tikrinama kiekviena klasės savybė po kiekvieno pokyčio. Tai galima pakeisti, kada nurodoma, kad modeliavimo įrankis vykdytų validacijos taisyklę tiktai pasikeitusiai klasės savybei. Tai galima atlikti sukuriant Java klasę, kuri papildo c*om.nomagic.magicdraw.validation.DefaultValidationRuleImpl* klasę naudojant atitinkamą metodą.

Šių metų vasarą buvo pristatyta nauja *MagicDraw* produkto 18.4 versija. Naujojoje versijoje patobulinti tokie dalykai kaip lengvesnis ir greitesnis būdas kurti ir demonstruoti modelius, išplėstos

Teamwork Cloud integracijos funkcijos, kurios palengvina komandos bendradarbiavimą. Naujas įskiepis palengvina analizę dėl projekto saugumo ir patikimumo. Su šiuo įskiepiu galima atlikti poveikio analizę (automatiškai identifikuoja silpnesnes būsenas, reikalaujančias dėmesio), validaciją (jeigu rizikos, adresuotos saugumo reikalavimų arba rizikos kontrolės priemonių) ir kita, tačiau modelio elementų validacijai jokių atnaujinimų nebuvo pateikta.

1.2.5. Validacijos įrankis *MagicDraw* programoje

MagicDraw programoje taisyklėms taikomi apribojimų tipai

MagicDraw programoje kiekvienai validacijos taisyklei gali būti pritaikytas tam tikras apribojimo tipas – kokiam programos elementui taikoma kuriama taisyklė. Taisyklė yra sumodeliuota kaip apribojimas ir turi tikslinę nuosavybę (angl. *target classifier property*). Nuosavybė apibrėžia, kokiam elementų tipui ši taisyklė taikoma. Taip atsiranda metalygio atskyrimas. Apribojimai, apibrėžti tam tikriems klasifikatoriams, yra įvertinami šių tam tikrų klasifikatorių atvejuose, kada atliekama validacija. Taip pat gali būti tikrinamos klasės subklasės.

Išskiriami 3 apribojimų tipai, kuriuos *MagicDraw* gali vertinti:

- Klasifikatoriaus lygio apribojimai. Tai apribojimai, kurie yra pateikti klasėse, duomenų tipuose ir kituose modelio klasifikatoriuose. Jie yra vertinami visuose klasifikatorių atvejuose.
- Apribojimai metaklasėms - tai kada apribojimas taikomas metaklasei (viena iš klasių - *UML Standart Profile::UML2 Metamodel*), apribojimas taikomas visiems to pačio tipo modelio elementams. Pavyzdžiui, jeigu apribojimas taikomas metaklasei *Aktorius*, tuomet šis apribojimas taikomas visiems klasės *Aktorius* elementams modelyje. Pateiktame taisyklės pavyzdyje aprašomas apribojimas, kuris nusako, kad visi aktorių vardai modelyje turi būti rašomi iš didžiosios raidės:

  ```
  context Actor inv capitalize:
  let startswith:String = name.substring(1,1) in
  startswith.toUpper() = startswith
  ```

  Tokie apribojimai yra naudingi apibrėžiant bendrines taisykles, kurios taikomos tam tikriems elementams visame modelyje.
- Apribojimai stereotipams - tai kada apribojimai taikomi stereotipams. Apribojimas taikomas visiems modelio elementams, kurie turi pritaikytą stereotipą. Šie apribojimai naudingi, kada kuriami dalykinės srities specifiniai profiliai. Adaptuojant *UML* į specifinio modeliavimo dalykinę sritį, profilis paprastai sukuriamas su išplėtimais tai dalykinei sričiai – stereotipais, žymėmis ir pan. Apribojimai stereotipams leidžia užtikrinti tam tikros dalykinės srities taisykles.

Iš anksto numatyti validacijos paketai

MagicDraw yra keli validacijos paketai su validacijos taisyklių rinkiniais [8]. Atsižvelgiant į tai, jog validacijos taisyklės ir jų rinkiniai yra įrankis, patalpintas modelyje, validacijai reikalingi validacijos paketų sąrašai priklauso nuo to, kokie profiliai sudaro modelį [9].

UML standartinis profilis sudarytas iš dviejų validacijos taisyklių paketų. Šie du paketai yra visuose modeliuose:

- UML išbaigtumo (angl. *completeness*) apribojimai;
- UML korektiškumo (angl. *correctness*) apribojimai.

Išbaigtumo paketas sudarytas iš taisyklių rinkinio, kuris tikrina, ar modelis yra išbaigtas, t. y. kad nebūtų atotrūkių, kad būtini informacijos laukai elementuose būtų užpildyti (pavyzdžiui, tikrina, ar visos ypatybės turi apibrėžtą tipą ir pan.).

Korektiškumo paketas sudarytas iš taisyklių rinkinio, kuris tikrina bendrines klaidas modeliuojant *UML2* kalba. Reikėtų atkreipti dėmesį, kad šis rinkinys nėra išsamus.

Taip pat kiekvienai iš modeliuojamų sričių – *XML* schemoms, *DDL*, *Java*, *C++*, *DoDAF*, *SysML* – yra atitinkami validacijos paketai. Šie paketai yra apibrėžti atitinkamuose modeliavimo sričių profiliuose, todėl jie įtraukiami automatiškai, kai pradedamas modeliavimas toje srityje. Pavyzdžiui, jeigu kuriama nauja *XML* schemos diagrama, tuomet *XML* schemos profilis autimatiškai įtraukiamas į modelį ir šis profilis įtraukia *XML* schemų validacijos paketą su savimi.

Validacijos taisyklių rinkiniai

Validacijos taisyklių rinkinius galima susikurti patiems, arba naudoti jau prieš tai aprašytus - sukurtus iš anksto, tokius kaip *UML* išbaigtumo ir korektiškumo. Validacijos taisyklių paketas apibrėžia validacijos taisyklių rinkinį, kuris taikomas atliekant validaciją. Validacijos taisyklių rinkinio tikslas – sugrupuoti apribojimus jų nekartojant.

Naują validacijos taisyklių rinkinį galima kurti per validacijos parinkčių skirtuką, kuriame yra pasirinkimas sukurti naują validacijos taisyklių paketą. Validacijos taisyklių skirtuke apibrėžiamos validacijos taisyklės.

Validacijos taisyklių rinkinys modelyje saugojamas kaip paketas, kuriam pritaikytas <*ValidationSuite*> stereotipas. Validacijos paketų lange pateikti visi tokie modelio paketai – taisyklių rinkiniai. Kitas alternatyvus būdas sukurti validacijos taisyklių rinkinį yra suteikiant paketui <*ValidationSuite*> stereotipą.

Kada sukuriamas apribojimas ir įtraukiamas į paketą, atitinkamas elemento įtraukimo ryšys sukuriamas iš modelio.

Taip pat gali būti apribojimų, kurie saugomi tiesiogiai validacijos rinkinio pakete – jie taip pat laikomi to rinkinio dalimi. Tokie apribojimai fiziškai egzistuoja pakete, todėl negali būti išmesti iš rinkinio. Paprastai validacijos taisyklės yra laikomos apribotame elemente, bet tam tikrais atvejais, kada apribotas elementas turi tik skaitymo funkciją, pavyzdžiui, patalpintas tik skaitymui skirtam profilyje, norint pridėti apribojimų reikia redaguoti profilį. Atskiras apribojimų grupavimas yra paprastesnis būdas.

Daugiau negu dviejų taisyklių rinkinių grupavimas į vieną ir dalijimasis

Norint supaprastinti modelio tikrinimą, kada reikalingas daugiau negu vienas validacijos taisyklių rinkinys, galima kelis taisyklių rinkinius sujungti į vieną. Ši funkcija atliekama per validacijos paketų parinktį, kurioje galima pasirinkti kurti naują taisyklių rinkinį – sukuriamas naujas paketas su nauju unikaliu pavadinimu. Tokiais naujai sukurtais validacijos taisyklių paketais galima dalintis ir perkelti į kitus modelius. Validacijos taisyklių ar jų rinkinių dalijimasis galimas per standartinį *MagicDraw* modulio įrenginį. Paketas su taisyklėmis gali būti išeksportuotas kaip modulis ir naudojamas kituose projektuose.

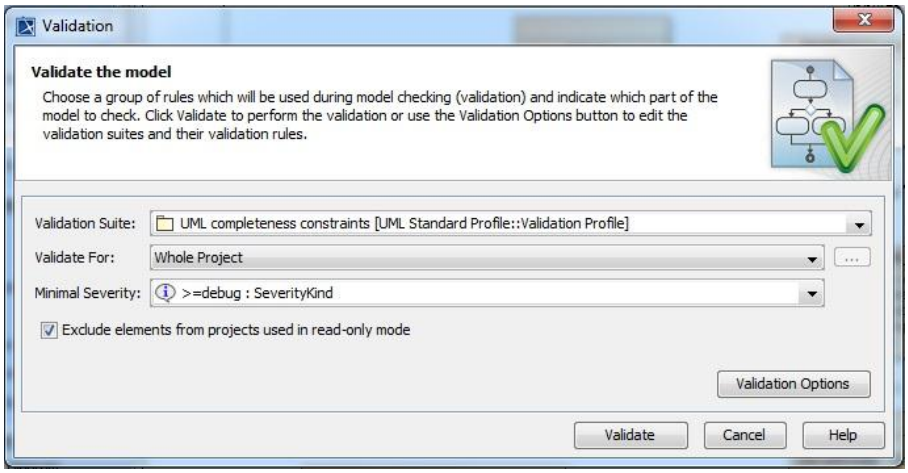
Kitas būdas dalintis taisyklėmis yra jų kopijavimas tarp projektų. Atsižvelgiant į tai, jog validacijos taisyklės yra tiesiog modelio elementai, su jais gali būti atliekami bet kokie veiksmai. Pavyzdžiui, pažymėjus taisyklių paketą su stereotipu <*ValidationSuite*>, iš paketo kontekstinio meniu pasirinkus moduliai ir eksportuoti modulį, pasirinktą paketą galima išsaugoti kaip atskirą elementą su *MagicDraw .mdzip* plėtiniu.

Turint išeksportuotą taisyklių rinkinį, jį lengvai gali įsidėti tiek kitas vartotojas į savo projektą, tiek galima naudoti kituose projektuose. Tai atlikti galima pasirinkus meniu punktą failas ir naudoti modulį. Atsidariusiame naudoti modulį dialogo lange pasirenkamas kelias iki norimo įsidėti taisyklių paketo. Taip pat papildomai galima nustatyti įkeliamo modulio importavimo nustatymus. Taigi naujai pridėtas modulis į projektą ir pridėto paketo apribojimų taisyklės gali būti naudojamos projekto validacijai.

Validacijos funkcija įrankyje

Norint atlikti validaciją, reikia pasirinkti validacijos taisykles ar jų paketą, kuris bus naudojamas, ir nurodyti, kuri modelio dalis ar visas modelis bus tikrinamas.

Norint atlikti *UML* modelio korektiškumo validaciją, reikia iš *Analyze* meniu pasirinkti *Validation* ir paspausti *Validate* mygtuką. Atsidariusiame *Validation* lange (žr. 1.10 pav.) *Validation Suite* vietoje reikia pasirinkti *UML* korektiškumo apribojimus. Visi galimi validacijos paketai surašyti šioje vietoje.



1.10 pav. Validacijos parinktys *MagicDraw* programoje

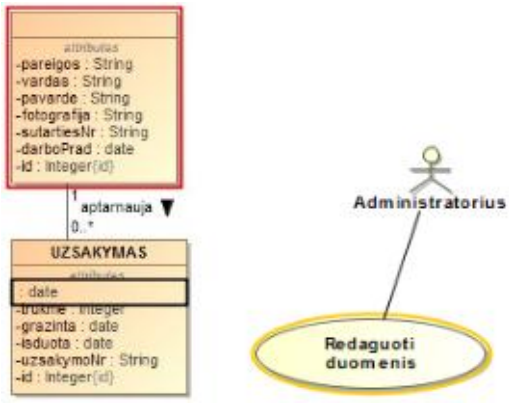
Priklausomai nuo atidaryto projekto, priklauso ir galimų validacijos paketų sąrašas. Validacijos paketai ir taisyklės yra patalpinti modelyje kaip įprasti modelio elementai. Pagal nutylėtus nustatymus, projektas turi prieš tai aprašytus paketus – *UML* išbaigtumo ir korektiškumo apribojimų, apibrėžtų standartiniame profilyje.

Jeigu projektas naudoja kitus profilius, tokius kaip *Java*/*XML schema*/*DDL* profiliai, šie profiliai įtraukia savo numatytuosius taisyklių paketus. Čia galima susikurti savo validacijos taisykles ir jas sugrupuoti į paketus, ir šis sukurtas paketas bus pasiekiamas šiame validacijos dialogo lange.

Atsidariusiame validacijos parinkčių lange *Validate For* vietoje reikia pasirinkti validacijos objektą, kuriuo gali būti arba visas projektas (angl. *whole project*), arba tam tikri paketai ir / ar elementai, kurie gali būti pasirenkami per dialogo lange pateikiamą daugtaškio mygtuką. Taip pat galima pasirinkti lygį, kokiu mastu bus tikrinami elementai. Derinimas (angl. *debug*) yra žemiausias galimas lygis, kuriame galima pasirinkti validacijos taisykles. Po šio pasirinkimo visos validacijos taisylės paleidžiamos modelio tikrinumui. Validacija visada yra rekursinė, taigi jei pasirenkamas taisyklių paketas validacijai, nereikia papildomai rinktis paketo vidinių elementų. Jeigu norima validuoti modulius, turinčius tik skaitymo (angl. *read-only*) tipą , arba elementus, kurie yra tuose moduliuose, reikia nuimti varnelę nuo pasirinkimo „*Exclude elements from read-only modules*“ (standartiškai varnelė yra uždėta). Atlikus validaciją, tikrinimo rezultatai pateikiami *Validation Results* skirtuke.

Po atliktos modelio validacijos atsidarius rezultatų skirtuką ir bet kurią modelio diagramą, galima matyti diagramos elementus ir ryšius, kurie pažeidžia bent vieną taisyklę ir yra įrašyti lentelėje. Priklausomai nuo taisyklės pažeidimo kritiškumo, elementai ar ryšiai žymimi atitinkamomis spalvomis (1.11 pav.):

- Klaida (angl. *error*) – raudonai;
- Įspėjimas (angl. *warning*) – geltonai;
- Suderinimo (angl. *debug*) – pilkai;
- Informacijos (angl. *info*) – juodai.

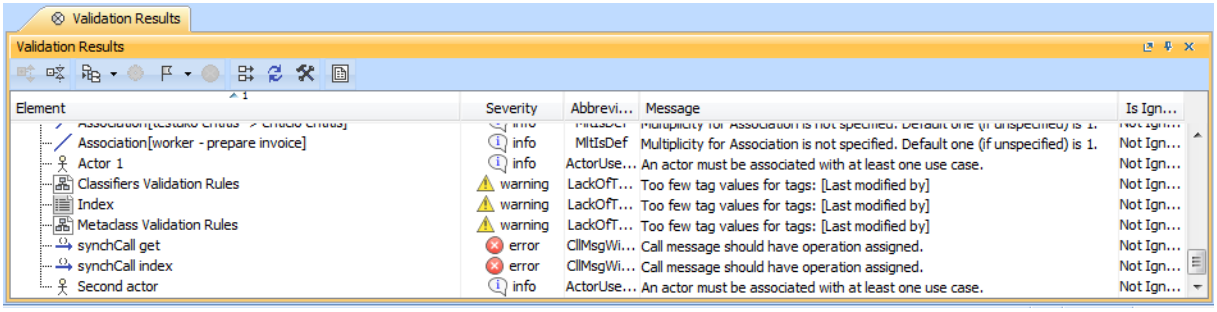


1.11 pav. Aptiktų klaidų žymėjimo pavyzdys

Jeigu elementas pažeidžia kelias taisykles, tuomet žymėjimo spalva priklauso nuo to, kokia rimčiausia taisyklė yra pažeista. Elementų žymėjimas spalvomis išlieka tol, kol ištaisoma pažeidimo klaida, arba uždaromas klaidų rezultatų skirtukas.

Validacijos rezultatų skirtukas

Validacijos rezultatų skirtuko pavyzdys pateikiamas 1.12 paveiksle.



1.12 pav. Validacijos rezultatų skirtuko pavyzdys

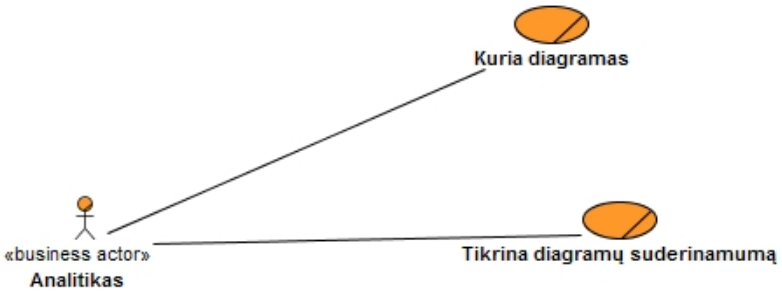
Validacijos rezultatų skirtukas atidaromas automatiškai iškart po to, kai tikrinimas baigiamas vykdyti.

Validacijos rezultatų skirtuke yra tokios skiltys:

- Elemento (angl. *element*) skiltyje pateikiami elementai, kurie pažeidžia apribojimo taisyklę.
- Kritiškumo (angl. *severity*) skiltyje pateikiamas taisyklės kritiškumo pažeidimo lygis.
- Santrumpos (angl. *abbreviation*) skiltyje pateikiamas pažeisto apribojimo trumpinys, kuris dažniausiai naudojamas rūšiuojant arba grupuojant.
- Žinutės (angl. *message*) skiltyje yra aprašomasis tekstas, kuris nusako pažeidimą.
- Ar ignoruojamas (angl. *is ignored*) skiltyje pateikiama indikacija, ar pažeidimas ignoruojamas ar ne.

1.3. UML modelio naudotojų analizė

Šiomis *UML* modelio suderinimo priemonėmis galės naudotis vartotojai, pavyzdžiui analitikai (žr. 1.13 pav.), kuriems reikia kurti *UML* diagramas. Analitikas ar bet kuris kitas darbuotojas, atsakingas už sistemos modeliavimą, pirmiausiai gali sukurti diagramas, tuomet gali tikrinti sukurtų diagramų suderinamumą.



1.13 pav. UML modelio naudotojų analizės schema

1.4. Esamų UML modelio elementų suderinimo problemos sprendimo metodų analizė

UML turi 13 diagramų, kurios atvaizduoja sistemą iš skirtingų pusių, skirtingais aspektais. Kiekviena diagrama apibūdina vis kitokią sistemos dalį, pavyzdžiui, panaudojimo atvejų diagramos aprašo, ką sistema turi daryti, o sekų diagramų pagalba nusakoma, kaip sistema iš tikrųjų atliks numatytas funkcijas. Tam tikra *UML* diagrama atvaizduoja tik dalį sistemos viso modelio ir dažniausiai ne visą, todėl labai patogu, kad modelio turinys yra padalijamas į diagramas, rodančias sistemą skirtingais pjūviais. Labai nesunku diagramose pakeisti modelį sudarančius elementus, ir taip atvaizduoti tik mažą sistemos dalį su tik tam modeliui reikalinga informacijos dalimi. Dažnai skirtingose diagramose pateikiama ta pati sistemoje esanti informacija, tik iš kito sistemos pjūvio taško, todėl labai svarbu užtikrinti, kad vienoje diagramoje naudojami elementai, atitinkantys tam tikrą sistemos dalį, kitoje diagramoje būtų naudojami taip pat tie patys elementai, nors ir diagrama formuojama iš kito požiūrio taško.

UML modelių suderinamumo tikrinimas yra aktuali ir dažnai tyrinėjama tema. Ištirta ir pristatyta daug skirtingų požiūrių šiuo klausimu. Patys pirmieji tyrinėjimai buvo atlikti 2002 metais mokslininko Liu ir kitų [10], kurie pasiūlė *UML* modelių tikrinimą, kai *UML* modeliai ir suderinamumo tikrinimo taisyklės konvertuojamos į formaliąją kalbą. Jie plačiai domėjosi modelių nesuderinamumu, įtraukiant problemas, susijusias su informacijos dubliavimusi, standartų ir reikalavimų neatitikimu, pasikeitimų plitimu, plėtojant kuriamą modelį. Jų tikslas buvo sukurti programinės įrangos projektavimo aplinką, kuri modeliuose pritaiko automatizuotą projektavimo neatitikimų aptikimą ir pritaiko sprendimą.

Mokslininkai Rash‘as ir Wehreim‘as [11] siūlė naudoti *Process Algebra*, *Object-Z* kalbas. Jie atliko tyrimus remdamiesi keliais suderinamumo aspektais, kurie paremti bendra formaliąja semantika ir taikomi tiriant tiek klasių, tiek būsenos diagramų suderinamumą. Taip pat buvo parodyta, kaip suderinamumo tikrinimas gali būti vykdomas automatiškai. Mokhatis [12] pasiūlė formalų *UML* diagramų tikrinimą naudojant *Maude* kalbą. Formalioji ir objektinė *Maude* kalba, paremta perrašant logiką, palaiko formalias specifikacijas ir modelių tikrinimą. Taikant sukurtas *Maude* specifikacijas, jos buvo patvirtintos atliekant simuliaciją ir *UML* klasių, būsenos ir komunikacijos diagramų tikrinimą. Miloudi [13] teikė pirmenybę Z kalbos taikymui formaliems modeliams. Jis taikė diagramų konvertavimą į kitą, šiuo atveju Z, kalbą. Buvo sukurtas įrankio prototipas, skirtas automatiškai generuoti analogišką formalią *UML* klasių diagramos kopiją, taip aptinkant nesuderinamumus.

Pagrindinė šių požiūrių nauda yra suderinamumo patikrinimo lengvumas. Nesuderinamumo aptikimo algoritmų prieinamumas ir jų pritaikymas projektavimo įrankiuose. Vienas iš šių požiūrių trūkumų, tai kad praktikoje formaliosios kalbos nėra populiarios kuriant modelius, nors jos ir yra ir tikslesnės, ir aiškesnės.

Kotulskis [14], Wang‘as [15] tobulino prieš tai aprašytų metodikų taikymą, tačiau tokių, kurios turi vaizdinę išraišką, pavyzdžiui, *NLC* (angl. *Node Label Controlled*), grafikus, *OWL-DL* (angl. *Ontology Web Language*). Kotulskis teigė, kad tiek *UML* diagramų modeliavimo proceso, tiek

projektuotojo sprendimų, formalus apibūdinimas sunkina jo struktūrą, ir kad paraleliai turi būti palaikomos abi veiklos, taip pat lokalių grafikų transformacijos sistemos ir jų bendradarbiavimas užtikrinant sistemos suderinamumą. Tuo tarpu Wang‘as sukūrė ontologija paremtą metodą semantikai tikrinti, kada statinė semantikos dalis algoritmais yra pakeičiama į *OWL-DL*, o dinaminė dalis apibūdinama *DL-Safe* taisyklėmis, ir taip įgalinamas modelio suderinamumo tikrinimas naudojant ontologijomis paremtą įrankį.

Pagrindinis šių tyrimų trūkumas yra tai, kad šie modeliai, taisyklės nėra pritaikomos *OMG* (angl. *Object Management Group*) sukurtam *UML* metamodeliui. Šie požiūriai yra susieti su *UML* modelių aprašymais, apibrėžtais Kotulski ir kitų tyrėjų. Taip pat *UML* modelių vertimas į formalius modelius reikalauja papildomų išteklių.

Kita *UML* modelių tikrinimo tyrimų grupė, kuri plėtojama praktiškai paraleliai su *UML* modelių vertimo į formalios kalbos modelius, yra apribojimais grindžiamas požiūris. Pagrindinė šių tyrimų idėja – siūlymas tikrinti pagal apibrėžtus apribojimus iš dalies formalius modelius. Jie sukuriami vartojant kalbą, kurios sintaksė apibrėžta formaliai, tačiau didžioji dalis semantikos yra apibūdinta naudojant natūralią kalbą [16]. Tyrimai šioje grupėje skiriasi tikrinamomis savybėmis (suderinamumas, korektiškumas) ir taisykles išreiškiančiamomis kalbomis. Chiorean‘as [17], Pakalnickienė [18], B. Hnatkowska [19] siūlo *UML* modelių korektiškumo tikrinimą pagal *OCL* taisykles, kurios apribotos vieno atžvilgio modeliu. Chiorean‘as teigia, kad naudojant *OCL* pagrindu sukurtą struktūrą, galima sukurti vertingą prieigą tikrinant *UML* modelius. Taip pat iš gautų rezultatų atliekant tyrimus pabrėžiami keli *UML* apibrėžimų trūkumai, ir įrodoma, kad *OCL* siūlo įrankių savybėms valdyti reikalingą pagalbą. Tuo tarpu Pakalnickienė pristato *OPCM* (angl. *Ordered and Precise Conceptual Model*) metodologiją, skirtą procesui tikrinti, kai abstraktūs modeliai yra tiriami siekiant užtikrinti apribojimus, kurie turi būti išpildyti kiekvienoje sistemos būsenoje. Chen‘as ir Motet [20] siūlo kontroliuoti gramatiką naudojant *C-Control* ir taip išreiškiant taisyklių korektiškumą. Naudojamą *C-System* sudaro kontroliuojama gramatika ir kontroliavimo gramatika. Gairės ir suderinamumo taisyklės yra formalizuojamos kaip kontroliavimo gramatika, kuri valdo *UML* panaudojimą. Sapna‘s ir Mohanty‘s [21] aprašo keletą *OCL* suderinamumo taisyklių pavyzdžių ir jų vertimą į *SQL*. Jų darbas koncentruojasi į intermodalinį lygį ir atkreipia dėmesį į struktūrinio suderinamumo problemą tarp panaudojimo atvejų, veiklos, bendradarbiavimo (angl. *collaboration*), būsenos ir klasės diagramų. Chanda [22] siūlo keletą laisvai išreikštų suderinamumo taisyklių. Jis siūlo taikyti formaliąją gramatiką trims dažniausiai naudojamoms *UML* diagramoms – panaudojimo atvejų, veiklos ir klasių. Buvo sukurta diagramų tikrinimo struktūra, kurią sudaro sintaksinis korektiškumas, suderinamumas tarp diagramų ir reikalavimų atsekamumas. Dubauskaitė [23] pasiūlė tikrinimo metodą, paremtą sukurtu *ConsistencyConstraints4UML* moduliu, kuris pagal keliamus reikalavimus sukuria UML modelių suderinamumo taisyklių rinkinį. Taisyklės yra apibrėžtos metamodelio lygyje ir gali būti realizuojamos bet kuriame *UML 2.2* metamodelį palaikančiame projektavimo įrankyje. Taip pat buvo pasiūlytas ir realiojo laiko modeliuojamų sistemų suderinamumo tikrinimas [24].

Pagrindinis tokių tiriamųjų darbų trūkumas tai, kad IS modelių suderinamumo tikrinimo algoritmai pateikiami neaiškiai ir netiksliai.

Kaip 1-oje lentelėje pavaizduota, dauguma metodų nagrinėja klasių diagramų suderinamumą. Miloudi ir Chiorean‘as nors atrodo kad nagrinėja tik klasių diagramas, iš tikrųjų jie aprašo ir kitų diagramų suderinamumą, tik taip ryškiai apie jas neužsimena, daugiausia atlieka tyrimus su klasių diagramomis. Tuo tarpu Wang‘as kartu su klasių diagramomis aprašinėja ir veiklos diagramas. Chen‘as ir Motet nagrinėja paketų diagramas, o Dubauskaitė – būsenų. Nors prie nei vieno iš šių metodų neminima daugiau diagramų, atliktų tyrimų aprašymuose galima įžvelgti, kad buvo naudojama ir kitų diagramų. Lentelėje minimos tik pagrindinės diagramos, su kuriomis buvo atliekami suderinamumo tyrimai.

Kaip prieš tai buvo minėta suderinamumo apraše, kiekvienas iš šios srities tyrinėtojų naudoja tam tikrą formalizavimo metodiką tyrimams atlikti. Miloudi‘s taikė diagramų konvertavimą į Z kalbą, Wang‘as sukūrė ontologija paremtą metodą semantikai tikrinti, kada statinė semantikos dalis algoritmais yra pakeičiama į *OWL-DL*, o dinaminė dalis apibūdinama *DL-Safe* taisyklėmis. Chen‘as ir Motet siūlo kontroliuoti gramatiką naudojant *C-System*, taip išreiškiant taisyklių

korektiškumą. Chiorean‘as ir Dubauskaitė siūlo *UML* modelių korektiškumo tikrinimą pagal *OCL* taisykles.

Taip pat galima išskirti dvi pagrindines metodų grupes, kada diagramos verčiamos į formalius modelius, arba taikomas apribojimais grindžiamas metodas. Du pirmieji metodai, Miloudi ir Wang‘as, taiko diagramų vertimo į formalius modelius metodą, kada diagramos pirmiau konvertuojamos į tam tikrą formaliąją kalbą, ir tik tada tikrinamas jų suderinamumas. Kitu atveju taikomas apribojimais grindžiamas metodas, kuris taikomas Chiorean‘o, Chen‘o ir Motet, Dubauskaitės tiriamuosiuose darbuose.

Dar vienas iš kriterijų yra tai, ar sukurtas įrankis tyrime aprašomai metodikai realizuoti. Šiuo klausimu teigiamai įvertinti galima tik Miloudi ir Dubauskaitę, kurie sukūrė tam tikrus įrankius savo tyrimuose aprašomoms metodikoms realizuoti. Tuo tarpu likę tyrinėtojai Wang‘as, Chiorean‘as, Chen‘as ir Motet savo darbuose nepateikė jokių užuominų apie tai, kad būtų sukūrę kokį nors metodikai realizuoti įrankį. Taip pat galima aptarti ir kokie įrankiai išvis buvo naudojami metodikoms realizuoti. Jau minėtas tyrėjas Miloudi straipsnyje nepateikia jokių užuominų apie tai, kokius įrankius naudojo atliekant savo tyrimus. Taip pat ir Wang‘as, kuris taip pat nepateikia informacijos apie tai, kokiais įrankiais naudojosi atlikdamas savo tyrimus. Prie šios tyrėjų grupės taip pat būtų galima priskirti ir Chen‘o su Motet metodo tyrimą, kuris taip pat traktuojamas kaip nenagrinėjęs jokių įrankių metodikai realizuoti, tačiau tai yra visiškai suprantama, nes šių mokslininkų tyrimai buvo atliekami tik teoriškai, ir jų tikslas nebuvo sukurtą metodiką pritaikyti tam tikrame įrankyje praktiškai. Taikymą jie numatė kitame savo tiriamajame darbe, o šis aprašomas remiantis tik teorinėmis žiniomis ir įžvalgomis. Tuo tarpu Chiorean‘as ir Dubauskaitė savo teorines įžvalgas pritaikė praktiškai įrankyje *MagicDraw*. Be to, Chiorean‘as pritaikė ne tik *MagicDraw*, bet ir *Rational Rose* įrankyje.

Dar vienas iš kriterijų lyginant metodus yra tai, ar buvo sukurta metodika, tikrinimo struktūra, tikrinanti suderinamumą. Visų tyrinėtojų, tiek Miloudi, Wang‘o, Chiorean‘o, Chen‘o ir Motet, Dubauskaitės turiamuosiuose darbuose buvo sukurta arba metodika, arba struktūra, kuri tikrina modelių ar diagramų suderinamumą.

Ir paskutinis kriterijus, pagal kurį buvo lyginami skirtingi metodai, yra ar visa tai pritaikoma *UML* metamodelio įrankiui. Chiorean‘o ir Dubauskaitės darbuose aprašoma, kad jų sukurtos metodikos yra pritaikomos *UML* metamodelio įrankiui. Tuo tarpu Miloudi, Wang‘o, Chen‘o ir Motet‘o tyrimuose apie tokį pritaikymą išvis neužsiminta, arba nenagrinėjama.

1 lentelė. Esamų modelio suderinimo problemos sprendimų palyginimas

Metodai / Kriterijai	Miloudi [13]	Wang [15]	Chiorean [17]	Chen ir Motet [20]	Dubauskaite [23]
Kokių diagramų suderinamumas nagrinėjamas?	Class	Activity Class	Class	Package Class	State Class
Kokia formalizavimo metodika naudojama?	Z kalba/notacija	OWL-DL DL-Safe taisyklės	OCL	C- System	OCL
Ar diagramos verčiamos į formalius modelius?	+	+	-	-	-

Metodai / Kriterijai	Miloudi [13]	Wang [15]	Chiorean [17]	Chen ir Motet [20]	Dubauskaite [23]
Ar tai apribojimais grindžiamas metodas?	-	-	+	+	+
Ar sukurtas įrankis metodikai realizuoti?	+	-	-	-	+
Kokie įrankiai naudoti metodikai realizuoti?	Nenagrinėjama	Nenagrinėjama	Rational Rose; *MagicDraw*	Nenagrinėjama (Apžvelgiama teoriškai)	*MagicDraw*
Ar sukurta metodika/tikrinimo struktūra, tikrinanti suderinamumą?	+	+	+	+	+
Ar pritaikoma UML metamodelio įrankiui?	Nenagrinėjama	Nenagrinėjama	+	Nenagrinėjama	+

1.5. Darbo tikslas, uždaviniai, planas ir siekiami privalumai

Šio darbo **tikslas** – palengvinti UML modelio kūrimą ir pagerinti modelio kokybę pasiūlant modelio elementų tarpusavio suderinimo priemones ir įrankius. Šiame darbe buvo išsikelti tokie **uždaviniai**:

1. Išanalizuoti UML kalbą ir diagramų tarpusavio ryšius.
2. Išanalizuoti metodus, naudojančius UML, pasirinkti metodą, kurio metu kuriamas diagramas būtų naudinga suderinti.
3. Išanalizuoti suderinimo priemones ir pasirinkti tinkamą.
4. Išanalizuoti įrankius, kuriuose būtų galima taikyti suderinimo priemones, ir pasirinkti tinkamą įrankį.
5. Pasiūlyti metodiką UML modelio suderinimui.
6. Pasiūlytą metodiką realizuoti pasirinktame įrankyje.
7. Išbandyti metodikos veikimą patobulintame įrankyje.
8. Eksperimentiškai ištirti metodikos taikymą praktikoje.

Šiame darbe sukurtas įrankis, kuris palengvina modelio elementų suderinimą. Dažnai kuriant diagramas įrankyje paliekama daug laisvės ir galima sukurti nekorektiškas diagramas, todėl reikalingas įrankis, kuris užtikrina tokio modelio diagramų elementų suderinamumą. Modeliuojant informacines sistemas naudojamos įvairios skirtingos programos, taip pat programose sistema gali būti modeliuojama pasirinkta kalba, kurių taip pat yra ne viena. Pasirinkus kalbą, kuriai taikomas įrankis, ir programą, kurioje įrankis realizuojamas, sukurtas diagramų elementų suderinamumo įrankis, užtikrinantis korektiškesnių diagramų kūrimą. Taip pat naudojantis šiuo įrankiu – sukurtu taisyklių rinkiniu - palengvinamas sistemos projektuotojo darbas, kuris lengviau ir greičiau gali modeliuoti sistemą.

Šiuo darbu siekiama sukurti UML įrankio patobulinimą, kuriuo naudojantis galima palengvinti tiek kuriamos sistemos modelio kūrimą, tiek modelio diagramų elementų suderinamumo užtikrinimą. Modeliuojant sistemą projektuotojui sudėtinga rankiniu būdu užtikrinti, kad visos modelio diagramos ir jų elementai tarpusavyje būtų suderinti, todėl sukūrus šį įrankį, kuris bent dalinai automatizuoja modelio elementų suderinamumo patikrinimą, projektuotojo darbas palengvinamas, o projektuojamos sistemos modelis tampa korektiškesnis ir labiau suderintas.

1.6. Analizės išvados

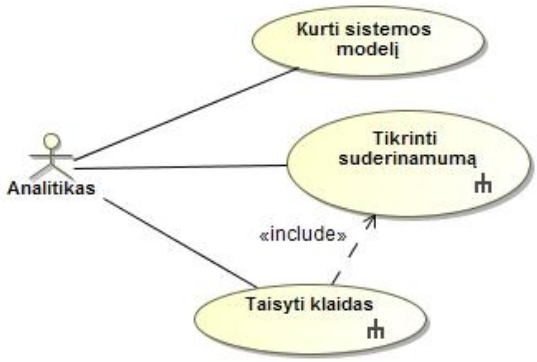
1. Atlikus analizę nustatyta, kad tie patys UML modelio elementai yra naudojami skirtingose diagramose ir įrankiai neužtikrina visiško elementų ir diagramų suderinamumo, todėl norint sugriežtinti elementų suderinamumą reikalingos taisyklės.
2. Išanalizavus mokslinius straipsnius UML modelio suderinimo srityje nustatyta, kad darbuose akcentuojami formalūs arba apribojimais grindžiami metodai, kurie nėra pritaikyti jokiam programinės įrangos kūrimo metodui, o šiame darbe sukurtas taisyklių rinkinys pritaikytas ICONIX metodu kuriamų modelių diagramų elementų suderinamumui tikrinti.

2. ICONIX METODU KURIAMO UML MODELIO ELEMENTŲ SUDERINIMO PROJEKTAS

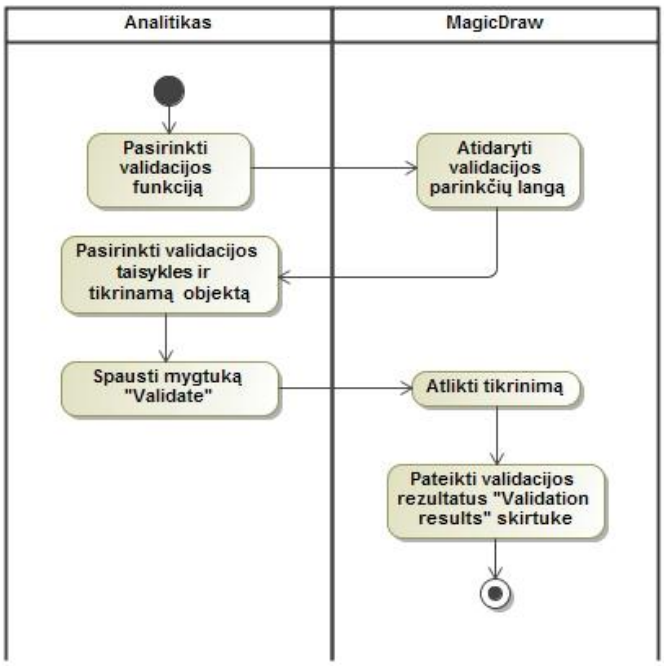
2.1. Validacijos taikymas *MagicDraw*

Kuriant didelius projektus naudojami UML modeliai, susidedantys iš daugelio tarpusavyje susijusių ir viena nuo kitos priklausomų skirtingų diagramų, kurios apibūdina skirtingus modelio aspektus. Tobulinant ir papildant projekto modelį, keičiamos ir diagramos, o šių pokyčiai gali daryti įtaką kitiems tos pačios diagramos elementams arba net kitoms susijusioms diagramoms, todėl diagramos tarpusavyje turi būti suderintos norint išvengti kuriamo projekto nesėkmių. Diagramų suderinamumas turi būti vienas svarbiausių užtikrinant nesuderinamumų efektyvų aptikimą, analizę ir patikslinimą. Taigi modelio neatitikimų aptikimas yra labai svarbus užtikrinant kuriamo projekto modelio korektiškumą.

2.1 pav. pateikiama, kokios yra asmens, kuris kuria sistemos modelį, funkcijos. Paveiksle pateikiamas naudotojas yra „Analitikas“, tačiau vietoj jo gali būti ir sistemos projektuotojas, ar bet kuris kitas asmuo, kuris modeliuoja kuriamą sistemą. Taigi vienas iš panaudojimo atvejų yra „Kurti sistemos modelį“, nes norint atlikti sekančius veiksmus, reikia turėti bent vieną diagramą. Panaudojimo atvejis „Tikrinti suderinamumą“ nusako, kad sukurtam modeliui atliekama validacija su pasirinktomis taisyklėmis, kuriomis norima patikrinti modelio suderinamumą, ar modelis tenkina atitinkamas taisykles ir sąlygas. *MagicDraw* yra jau iš anksto sukurti ir numatyti validacijos paketai ir taisyklių rinkiniai. Šis *MagicDraw* funcionalumas aprašytas kituose poskyriuose. Taip pat galima sukurtas taisykles grupuoti į savo kuriamus paketus. Atlikus modelio tikrinimą su pasirinktomis taisyklėmis, validacijos rezultatai pateikiami validacijos rezultatų skirtuke. „Tikrinti suderinamumą“ panaudojimo atvejo veiklos diagrama pateikiama 2.2 paveiksle.

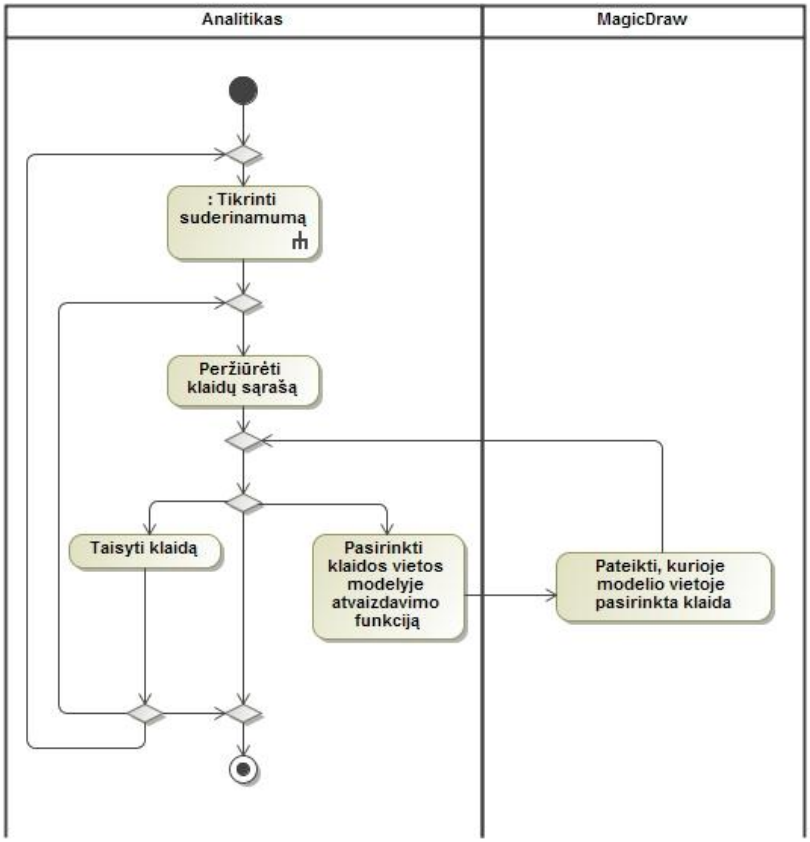


2.1 pav. Analitiko atliekamos funkcijos *MagicDraw* programoje



2.2 pav. Veiklos diagrama „Tikrinti suderinamumą“

Dar viena analitiko funkcija yra „Taisyti klaidas“, kurią atlikdamas analitikas gali ištaisyti validacijos įrankio aptiktas klaidas. Patikrinus modelio ar elementų suderinamumą – priklausomai, kas buvo pasirinkta tikrintojo – po to galima peržiūrėti *MagicDraw* pateiktą klaidų sąrašą, taip pat pasirinkti, ar taisyti klaidas, ar uždaryti klaidų langą, ar pasirinkti atitinkamos klaidos vietos atvaizdavimą, kada *MagicDraw* parodo atitinkamos klaidos vietą sukurtame modelyje. Šio „Taisyti klaidas“ panaudojimo atvejo veiklos diagrama pateikiama 2.3 paveiksle.



2.3 pav. Veiklos diagrama „Taisyti klaidas“

MagicDraw programa turi funkcionalumą, kuris tikrina sukurtus modelius. Jis susideda iš:

- validacijos taisyklių rinkinio. Kiekviena validacijos taisyklė užfiksuoja dalį būtinų taisyklių, kurios turi patikrinti modelį. Validacijos taisyklės yra apibrėžiamos kaip nekintami apribojimai modelyje;
- vieno ar daugiau validacijos taisyklių komplekto. Kiekvienas paketas yra sudarytas iš grupės panašios prasmės validacijos taisyklių, taip taikant iškart visą taisyklių rinkinį.

Taikant validacijos taisykles galima tikrinti tiek visą modelį, tiek tam tikras modelio dalis. Kada vykdomas modelio tikrinimas, kiekviena iš taikomų taisyklių yra pritaikoma kiekvienam tinkamam vertinti elementui tikrinimoje modelio apimtyje. Kiekvienas elementas, kuris pažeidžia validacijos taisyklę, įrašomas į atitinkamą lentelę, kuri pateikiama baigus visą modelio validaciją.

Atsižvelgiant į tai, kad validacijos taisyklės ir jų paketai yra modelio elementai, šie gali būti naudojami standartiškai kaip ir *MagicDraw* modeliavimo įrankiai – gali būti kopijuojami, perkeliami ar redaguojami modelyje. Taip pat jie gali būti transformuojami į modulius, taip palengvinant perpanaudojamumą kituose projektuose, taip pat gali būti talpinami serveryje norint keistis informacija ir pan. Taip pat tai leidžia redaguoti iš anksto nustatytas taisykles ir apibrėžti naujas, reikalingas taisykles modeliams ar profiliams.

Kiekviena suderinamumo taisyklė yra glaustai ir aiškiai apibūdinama natūralia kalba, pavydžiui, privati operacija negali būti iškviesta (sekų diagramoje) operacijos, kuri priklauso kitai klasei. Vėliau taisyklės formalizuojamos naudojant OCL. Kiekviena iš taisyklių yra išreikšta Bulio logika (angl. *boolean expression*). Taip pat gali būti naudojami suderinamumo įspėjimai, kurie nurodo suderinamumo sąlygas, kurios turi būti patikrintos, pavyzdžiui, visos klasės operacijų prieš sąlygos negali pažeisti klasės nekintamumo (angl. *invariant*). Šie įspėjimai naudojami tam, kad iš kuriamų modelių būtų išgaunama reikalinga informacija, kuri naudojama sudarant suderinamumo taisykles.

Kiekviena poveikio analizės taisyklė yra OCL (angl. *Object Constraint Language*) sukurta specifikacija, skirta išvesti keletą elementų rinkinių, kurie atitinkamai gali būti skirtingų tipų elementai (pvz., klasės, operacijos), o taip pat ir potencialiai gali būti paveikti tam tikrų pokyčių (pvz., pridėtos žinutės). Struktūrinėje klasifikacijoje (angl. *taxonomy*) yra po atskirą poveikio analizės taisyklę kiekvienam pokyčių tipui. Pokyčių analizės taisyklės yra parašytos struktūriškai, tiksliai ir apibrėžtai tam, kad jas būtų galima lengvai peržiūrėti, patikslinti ar pakeisti besikeičiant ir visam UML standartui.

Validacijos taisyklių kūrimas yra modeliuojamas kaip *UML2* apribojimai. Kaip ir su validacijos taisyklių paketais, toks požiūris leidžia su validacijos taisyklėmis elgtis kaip su paprastais modelio elementais. Jie gali būti valdomi naudojant įprastus modeliavimo įrankius. Tokie elementai gali būti kopijuojami, perkeliami iš vienos modelio vietos į kitą, perdaromi į kitą atskirą modulį, patalpinami serveryje lengvesniam informacijos apsikeitimui ir pan.

Atsižvelgiant į tai, kad apribojimai gali turėti skirtingas semantines prasmes *UML2* kalboje, modeliuojant validacijos taisykles naudojamas specialus apribojimų nekintamumo (angl. *invariant*) tipas. Taigi norint išskirti šiuos apribojimus iš kitų apribojimų tipų, turi būti pritaikytas <*invariant*> stereotipas.

Taip pat validacijos taisyklės reikalauja tokios informacijos, kaip klaidos rimtumo (angl. *severity*) lygis, klaidos santrumpa ir klaidos žinutė, pateikianti pilną klaidos apibūdinimą. Visa ši informacija pateikiama validacijos rezultatų skirtuke.

Norint saugoti tokią informaciją, turi būti naudojamas specialus stereotipas <*validationRule*>, pareikalautas iš <*invariant*>. Jeigu norima naudoti apribojimą kaip validacijos taisyklę modelio tikrinimui, tuomet turi būti naudojamas formalesnis stereotipas (t.y. <*validationRule*>). Jeigu apribojimas kuriamas tik dokumentacijos tikslais ir neplanuojama apribojimo naudoti kaip validacijos taisyklės modelio tikrinimui, tuomet paprastas <*constraint*> stereotipas yra pakankamas.

Validacijos taisyklės gali būti patalpintos bet kurioje modelio vietoje, kur galima talpinti *UML2* apribojimus, tačiau paprastai tai būna klasifikatorius, kuris yra apribotas – klasės, duomenų

tipai ir pan. klasifikatoriaus lygio apribojimams, ir stereotipai metaklasifikatoriaus lygio apribojimams.

Norint sukurti validacijos taisyklę metaklasei reikia pasirinkti paketą, kuriame norima taisyklę saugoti ir paspaudus dešinį pelės mygtuką pasirinkti kurti naują elementą.

Norint sukurti validacijos taisyklę klasifikatoriui arba stereotipui reikia atsidaryti klasifikatoriaus arba stereotipo specifikacijų langą, rinktis „*Constraints*“, esantį kairiajame dialogo lango šone ir kurti mygtuką dešinėje dialogo lango pusėje ir tuomet pasirinkti *Constraint*.

Kuriant taisyklę jai reikia priskirti validacijos klaidos (angl. *severity*) lygį:

- Klaida (angl. *error*) –žinutė įprastiems neatitikimams;
- Įspėjimas (angl. *warning*) – naudojamas ne tokiems rimtiems neatitikimams kaip klaida, tačiau iš kurio vėliau gali kilti jau klaida;
- Suderinimo (angl. *debug*) – neatitikimo lygmuo, kuriam priskiriamos tos validacijos taisyklės, kurios neatitinka informacijos lygio apibūdinimo;
- Informacijos (angl. *info*) – situacijos, kurios gali būti vartotojui įdomios.
- Lemtingos (angl. *fatal*) – naudojama klaidoms, kurios iškraipo modelį arba neatitinka UML metamodelio struktūros. Šis lygis naudojamas itin retais atvejais, nes *MagicDraw* automatiškai užtikrina tokių klaidų prevenciją.

Validacijos išraiška yra patalpinta kaip *Specification* savybės reikšmė. UML2 išraiška turi dvi savybes – kalbą (angl. *language*) ir kūną (angl. *body*).

MagicDraw palaiko tokias išraiškos kalbas:

- OCL2.0 naudojama validacijos taisyklėms, aprašytoms OCL kalba;
- Dvejetainė (angl. *binary*) naudojama pažangesnėms ir labiau išplėstoms išraiškoms, kurios nėra taip lengvai aprašomos OCL kalba. Tokios išraiškos parašytos Java kalba, parengtos ir apibrėžtos *MagicDraw classpath*.
- Skriptai arba scenarijai (angl. *scrips*), tokie kaip *JavaScript*, *Jython*, *Jruby*, *Groovy*, *BeanShell*.
- Struktūruotos išraiškos (angl. *structured expression*).

Aprašomos taisyklės sintaksė tikrinama automatiškai *MagicDraw*, tačiau ne visada pavyksta aptikti sintaksės neatitikimų. Kada validacijos taisyklė paleidžiama tikrinimui, taip pat vyksta papildomi pačios taisylės sintaksės tikrinimai, tokie kaip atitinkamų savybių egzistavimas, tipų tikrinimai ir pan. taip užtikrinant, kad taisyklės išraiška būtų teisingai įvertinta. Paprastai *MagicDraw* viduje pirmiausiai generuoja *Java* kodą iš aprašytos taisyklės išraiškos ir tik patikrinus taisyklę viduje, taisyklė parengiama modelio validacijos vykdymui.

2.1.1. Siūlomas taisyklių rinkinys

Atitinkamai pagal ankstesniame skyriuje aprašytus metamodelius, kuriamos taisyklės modeliams tikrinti. Kuriamos dvi taisyklių grupės – viena grupė, kurioje taisyklės orientuotos į *ICONIX* metodo kūrimą, kitos taisyklės – taisyklės, kurios svarbios norint užtikrinti modelio suderinamumą, tačiau jos nėra aprašytos numatytuose *MagicDraw* validacijos paketuose. Taisyklės, užtikrinančios kuriamų diagramų ir jų elementų korektišką naudojimą, atsižvelgiant į *ICONIX* metodo etapus, pateikiamos 2-oje lentelėje.

ICONIX metodui pritaikytos taisyklės:

1. Kiekvienam panaudojimo atvejui sudaroma detalizuota išbaigtumo diagrama.
2. Klasės su ribiniu (angl. *boundary*) stereotipu negali turėti ryšio su kitomis ribinio stereotipo klasėmis išbaigtumo diagramoje.
3. Klasės su ribiniu (angl. *boundary*) stereotipu negali turėti ryšio su esybės (angl. *entity*) stereotipo klasėmis išbaigtumo diagramoje.

4. Klasės su esybės (angl. *entity*) stereotipu negali turėti ryšio su kitomis esybės stereotipo klasėmis išbaigtumo diagramoje.
5. Aktoriai negali turėti ryšio su esybės (angl. *entity*) stereotipo klasėmis išbaigtumo diagramoje.
6. Aktoriai negali turėti ryšio su valdiklio (angl. *control*) stereotipo klasėmis išbaigtumo diagramoje.
7. Kiekvienam panaudojimo atvejui sudaroma detalizuota sekų diagrama.
8. Kiekvienas išbaigtumo diagramos ribinės (angl. *boundary*) klasės egzempliorius turi būti panaudotas kaip sekų diagramos gyvavimo linija (angl. *lifeline*).
9. Kiekvienas išbaigtumo diagramos esybės (angl. *entity*) klasės egzempliorius turi būti panaudotas kaip sekų diagramos gyvavimo linija (angl. *lifeline*).

Kitos taisyklės:

1. Kiekviena operacija, įtraukta į sekų diagramos pranešimą, turi būti apibrėžta klasių diagramoje.
2. Kiekviena sekų diagramos gyvavimo linija (angl. *lifeline*) turi būti klasės egzempliorius (angl. *class*) klasės diagramoje.
3. Kiekvienai klasei, kurioje operacijos naudoja kitą klasę, privalo būti priklausomybės ryšys tarp šių klasių klasių diagramoje.

2 lentelė. Suderinamumo taisyklių priskyrimas ICONIX metodo etapui

Nr.	Etapas / Taisyklė	Reikalavimų	Preliminaraus planavimo	Detalizuoto projektavimo	Vykdymo
1.	Kiekvienam panaudojimo atvejui sudaroma detalizuota išbaigtumo diagrama.		+		
2.	Klasės su ribiniu (angl. boundary) stereotipu negali turėti ryšio su kitomis ribinio stereotipo klasėmis išbaigtumo diagramoje.		+		
3.	Klasės su ribiniu (angl. boundary) stereotipu negali turėti ryšio su esybės (angl. entity) stereotipo klasėmis išbaigtumo diagramoje.		+		
4.	Klasės su esybės (angl. entity) stereotipu negali turėti ryšio su kitomis esybės stereotipo klasėmis išbaigtumo diagramoje.		+		

Nr.	Etapas / Taisyklė	Reikalavimų	Preliminaraus planavimo	Detalizuoto projektavimo	Vykdymo
5.	Aktoriai negali turėti ryšio su esybės (angl. entity) stereotipo klasėmis išbaigtumo diagramoje.		+		
6.	Aktoriai negali turėti ryšio su valdiklio (angl. control) stereotipo klasėmis išbaigtumo diagramoje.		+		
7.	Kiekvienam panaudojimo atvejui sudaroma detalizuota sekų diagrama.			+	
8.	Kiekvienas išbaigtumo diagramos ribinės (angl. boundary) klasės egzempliorius turi būti panaudotas kaip sekų diagramos gyvavimo linija (angl. lifeline).			+	
9.	Kiekvienas išbaigtumo diagramos esybės (angl. entity) klasės egzempliorius turi būti panaudotas kaip sekų diagramos gyvavimo linija (angl. lifeline).			+	
10.	Kiekviena operacija, įtraukta į sekų diagramos pranešimą, turi būti apibrėžta klasių diagramoje.			+	
11.	Kiekviena sekų diagramos gyvavimo linija (angl. lifeline) turi būti klasės egzempliorius (angl. class) klasės diagramoje.			+	
12.	Kiekvienai klasei, kurioje operacijos naudoja kitą klasę, privalo būti priklausomybės ryšys tarp šių klasių klasių diagramoje.			+	

Reikalavimų etape apibrėžiama, kokias funkcijas sistema atliks. Šiame etape projekto nariai arba klientai aprašo funkcinius reikalavimus. Kuriamas dalykinės srities modelis – apibrėžiami

dalykinės srities terminai. Apibrėžiamas vartotojo ir sistemos sąveikos modelis. Pradedama nuo vartotojo sąsajos prototipo kūrimo ir nustatomi panaudojimo atvejai. Reikalavimų etape kuriamos diagramos – panaudojimo atvejų, klasių diagramos. Šiam etapui nėra pritaikytos nei vienos taisyklės, nes kol kas kuriamos tik dvi diagramos, kuriose neatskleidžiama jokia *ICONIX* metodo specifika. Šiam etapui galima taikyti pagal nutylėjimą *MagicDraw* programoje pateikiamas validacijos taisykles.

Preliminaraus projektavimo etape kuriamos išbaigtumo diagramos remiantis panaudojimo atvejais. Taip pat šiame etape atnaujinamas dalykinės srities modelis, kada rašomi panaudojimo atvejai ir sudaromos išbaigtumo diagramos. Nustatomos trūkstamos klasės. Preliminaraus planavimo etape naudojamos reikalavimų etape sukurtos panaudojimo atvejų ir klasių diagramos, taip pat sukuriamos naujos – išbaigtumo diagramos. Šiam etapui galima pritaikyti antrajam etapui numatytas validacijos taisykles.

Detalizuoto projektavimo etape kiekvienam panaudojimo atvejui kuriama po sekų diagramą detalizuotai parodant, kaip bus įvykdyti panaudojimo atvejai. Taip pat atnaujinamas dalykinės srities modelis, pridedamos operacijos dalykinės srities objektams. Dalykinės srities modelis tampa statiniu modeliu, arba klasės diagrama. Detalizuoto projektavimo etape naudojamo tos pačios diagramos iš prieš tai buvusio etapo, taip pat kuriamos naujos sekų diagramos.

Vykdymo etape rašomi vienetų testai ir programinis kodas. Atliekamas integravimas ir scenarijaus testavimas. Atliekama kodo priežiūra ir modelio atnaujinimas ruošiantis kitam vystymo darbų etapui.

2.2. ICONIX metodu kuriamo UML modelio reikalavimų apibendrinimas

1. Darbe pasiūlytas taisyklių rinkinys, apimantis dviejų diagramų tipų taisykles: sekų ir klasių, skirtas UML elementų ir jų ryšių tikrinimui, taip pat ir specifiniam ICONIX metodui pritaikytas taisykles, apimančias išbaigtumo diagramas, jų elementų ryšius.
2. Pasiūlytas taisyklių rinkinys, apimantis ir ICONIX metodu kurtų modelių validaciją, realizuotas kaip *MagicDraw* įrankio validacijos modulio dalis, kuri papildo standartines *MagicDraw* programos validacijos taisyklių dalis.

3. TAISYKLIŲ RINKINIO SPRENDIMO REALIZACIJOS PROJEKTAS

3.1. Taisyklių rinkinio realizacijos ir veikimo aprašas

Atlikus analizę ir numačius, kokios taisyklės būtų aktualios tiek UML modelio, tiek *ICONIX* metodu kuriamo projekto suderinimui, pateikta 12-a taisyklių, užtikrinančių kuriamų diagramų ir jų elementų korektišką naudojimą, atsižvelgiant į *ICONIX* metodo etapus.

Buvo realizuotos tokios taisyklės:

1. Kiekvienam panaudojimo atvejui sudaroma detalizuota sekų diagrama.
2. Kiekvienam panaudojimo atvejui sudaroma išbaigtumo (angl. *robustness*) diagrama.
3. Kiekvienas išbaigtumo diagramos ribinės (angl. *boundary*) klasės egzempliorius turi būti panaudotas kaip sekų diagramos gyvavimo linija (angl. *lifeline*).
4. Kiekvienas išbaigtumo diagramos esybės (angl. *entity*) klasės egzempliorius turi būti panaudotas kaip sekų diagramos gyvavimo linija (angl. *lifeline*).
5. Kiekviena operacija, įtraukta į sekų diagramos pranešimą, turi būti apibrėžta klasių diagramoje.
6. Kiekviena sekų diagramos gyvavimo linija (angl. *lifeline*) turi būti klasės egzempliorius (angl. *class*) klasės diagramoje.
7. Kiekvienai klasei, kurioje operacijos naudoja kitą klasę, privalo būti priklausomybės ryšys tarp šių klasių klasių diagramoje.
8. Klasės su ribiniu (angl. *boundary*) stereotipu negali turėti ryšio su kitomis ribinėmis klasėmis išbaigtumo diagramoje.
9. Klasės su esybės (angl. *entity*) stereotipu negali turėti ryšio su kitomis esybių klasėmis išbaigtumo diagramoje.
10. Klasės su ribiniu (angl. *boundary*) stereotipu negali turėti ryšio su kitomis esybių (angl. *entity*) klasėmis išbaigtumo diagramoje.
11. Aktoriaus klasės negali turėti ryšio su esybės (angl. *entity*) stereotipo klasėmis išbaigtumo diagramoje.
12. Aktoriaus klasės negali turėti ryšio su ribinio (angl. *boundary*) stereotipo klasėmis išbaigtumo diagramoje.

Toliau bus detaliau aptariama kiekviena pateikta taisyklė, detalizuojamas jos OCL aprašas, pateikiami *MagicDraw* programos ekrano vaizdai, kaip atrodo programoje suveikusi taisyklė. Taip pat pateikiami UML metamodelio fragmentai, kuriuose vaizduojamos metaklasės, naudojamos aptariamoje taisyklėje.

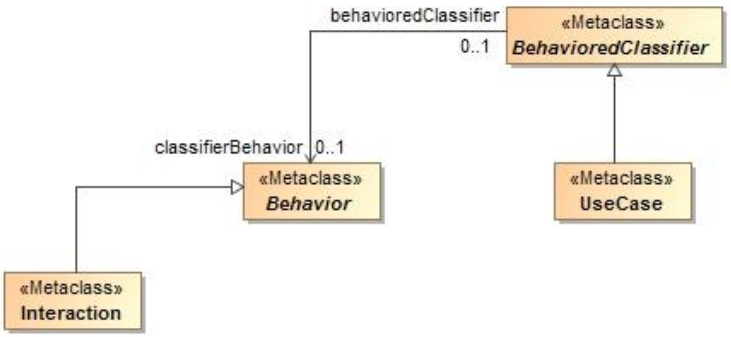
1. Kiekvienam panaudojimo atvejui sudaroma detalizuota sekų diagrama. Ši OCL kalba aprašyta taisyklė realizuota taip:

context UseCase inv UsecaseHasSequence:

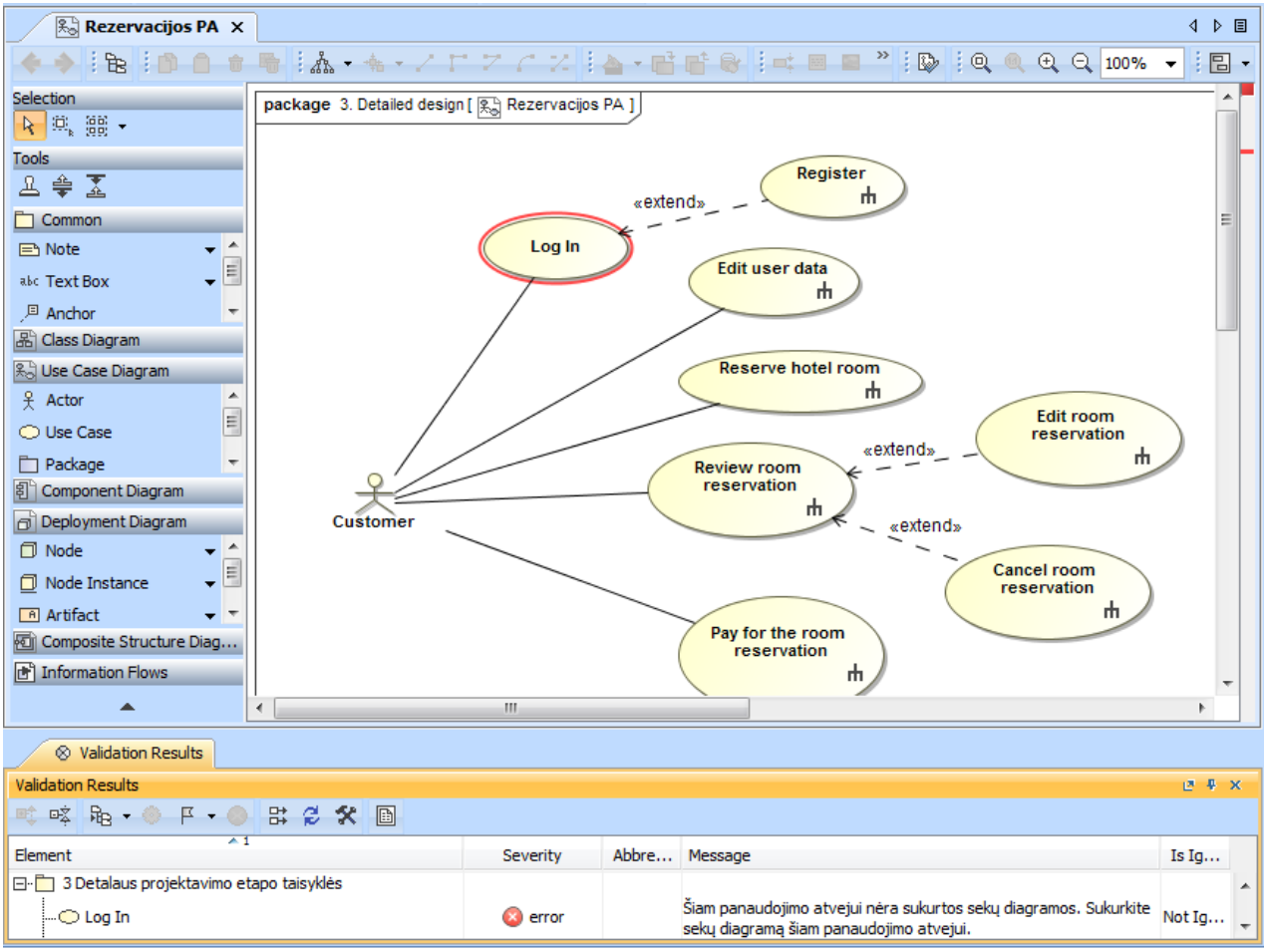
if self.classifierBehavior.oclIsUndefined() then false else (let diagrams:Collection(Diagram)=self.classifierBehavior.ownedDiagram in diagrams->size() > 0 and self.classifierBehavior.role->size() > 0) endif

Ši taisyklė tikrina, ar kiekvienas panaudojimo atvejis panaudojimo atvejų diagramoje turi po sekų diagramą. Pirmiausia taisykle patikrinama, ar panaudojimo atvejo parametras *classifierBehavior* nėra tuščias. Jeigu sąlyga teisinga, tada tikrinama kita taisyklės sąlyga, kuri naudoja *let* išraišką. Čia sukuriamas laikinas kintamasis *diagrams*, kuriame aprašomas diagramų *Diagram* rinkinys, kuriame tikrinama, ar egzistuoja nors viena diagrama tikrinamam panaudojimo

atvejui. Jeigu diagrama egzistuoja, tada taisyklė tikrina, ar panaudojimo atvejis turi parametrą *role*, kuris būdingas tik sekų diagramoms. Taip nustatoma, ar panaudojimo atvejis turi sekų diagramą. Paveiksle 3.1 pateikiamas metaklasių modelis, kuriame vaizduojama, kokios metaklasės buvo naudojamos kuriant pirmąją taisyklę. Paveiksle 3.2 pateiktas programos vaizdas, kaip atrodo programos langas, kai suveikia pirmoji taisyklė, kai padaroma klaida kuriant projektą *MagicDraw* programoje.



3.1 pav. Pirmojoje taisyklėje naudojamos metaklasės



3.2 pav. Pirmosios taisyklės taikymo pavyzdys

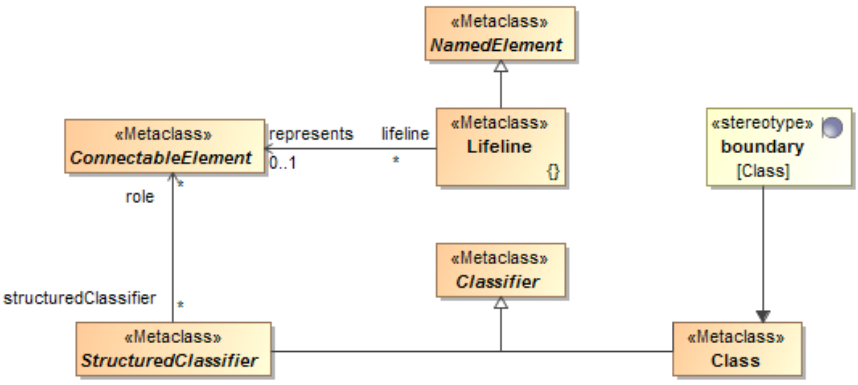
2. Kiekvienas išbaigtumo diagramos ribinės (angl. *boundary*) klasės egzempliorius turi būti panaudotas kaip sekų diagramos gyvavimo linija (angl. *lifeline*). Ši OCL kalba aprašyta taisyklė realizuota taip:

context Class inv BoundaryAsLifeline:

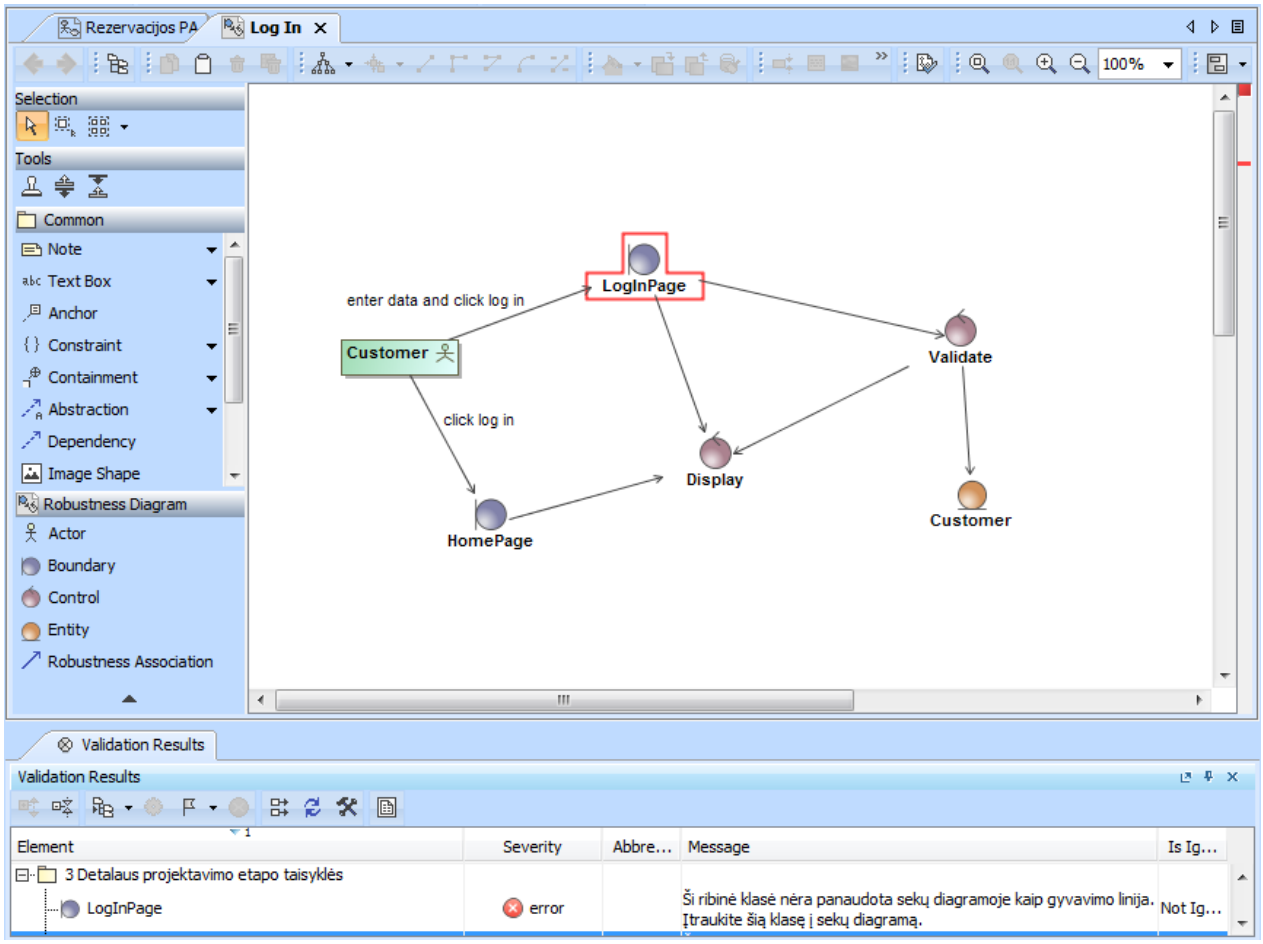
let names:Collection(String) = self.appliedStereotypeInstance.classifier->collect(name) in names->includes('boundary') implies

(let life:Collection(Type) = Lifeline::allInstances()->collect(v : Lifeline | v.represents.type) in life->includes(self))

Kuriant šią taisyklę naudojama *let* išraiša, kuria aprašoma sudėtinė sąlyga. Tai, kas aprašoma po išraiškos *let*, galima apibrėžti vietinės reikšmės kintamąjį, kuris gali būti naudojamas vietoj aprašytos sąlygos arba išraiškos. Šiuo atveju sukuriamas laikinas kintamasis *names*, kuriam priskiriamas *String* tipo masyvas, aprašomas išraiška *Collection*(*String*). Į masyvą surenkami elementai iš *class* metaklasės, kuriai taikoma ši taisyklė, pagal vardo *name* kriterijų. Taip pat nurodoma, kad surenkami elementai privalo turėti *boundary* stereotipą. Taip aprašyta pirmoji taisyklės sąlyga, kuri turi kartu būti korektiška ir grąžinti teisingą *true* rezultatą tam, kad visa taisyklė būtų korektiška ir veiktų pagal aprašymą. Antroji taisyklės sąlyga taip pat naudoja *let* išraišką. Šioje taisyklės dalyje tikrinama, ar sekų diagramos gyvavimo linija yra panaudota iš klasių diagramos. Čia taip pat nurodomas laikinas kintamasis *life*, kuriame surenkamas *Type* tipo masyvas, aprašomas rinkinyje *Collection*(*Type*). Šiam masyvui priskiriami visų *Lifeline* metaklasių objektų parametro *represents* tipai. Toliau tikrinama, ar laikiname kintamajame *life* yra tikrinamasis objektas. Jeigu yra, tuomet teigiama, kad taisyklė validi, išbaigtumo diagramos ribinių klasių egzemplioriai panaudoti kaip sekų diagramos gyvavimo linijos. Paveiksle 3.3 pateikiamas metaklasių modelis, kuriame vaizduojama, kokios metaklasės buvo naudojamos kuriant antrąją taisyklę. Paveiksle 3.4 pateiktas programos vaizdas, kaip atrodo programos langas, kai suveikia antroji taisyklė, kai padaroma klaida kuriant projektą *MagicDraw* programoje.



3.3 pav. Antrojoje taisyklėje naudojamos metaklasės



3.4 pav. Antrosios taisyklės taikymo pavyzdys

3. Kiekvienas išbaigtumo diagramos esybės (angl. *entity*) klasės egzempliorius turi būti panaudotas kaip sekų diagramos gyvavimo linija (angl. *lifeline*). Ši OCL kalba aprašyta taisyklė realizuota taip:

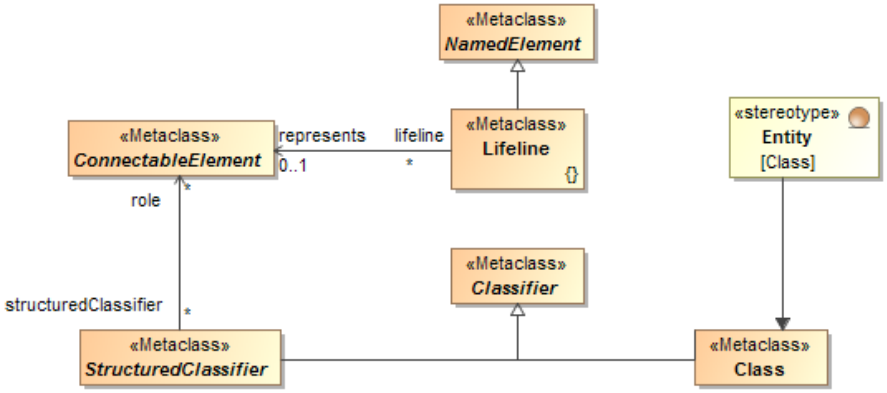
context Class inv EntityAsLifeline:

let names:Collection(String) = self.appliedStereotypeInstance.classifier->collect(name) in names->includes('Entity') implies

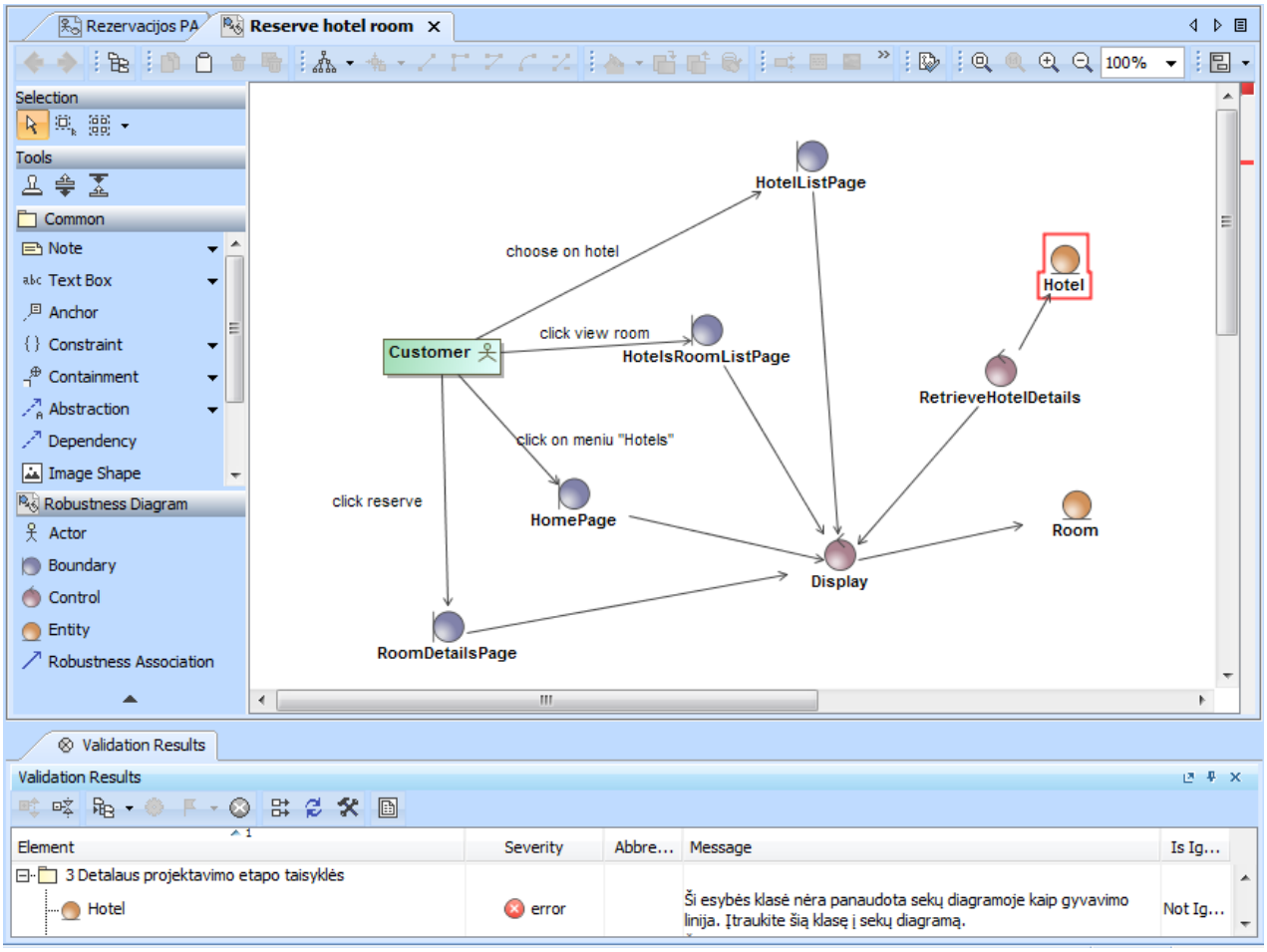
(let life:Collection(Type) = Lifeline::allInstances()->collect(v : Lifeline | v.represents.type) in life->includes(self))

Kuriant šią taisyklę naudojama *let* išraiša, kuria aprašoma sudėtinė sąlyga. Tai, kas aprašoma po išraiškos *let*, galima apibrėžti vietinės reikšmės kintamąjį, kuris gali būti naudojamas vietoj aprašytos sąlygos arba išraiškos. Šiuo atveju sukuriamas laikinas kintamasis *names*, kuriam priskiriamas *String* tipo masyvas, aprašomas išraiška *Collection*(*String*). Į masyvą surenkami elementai iš *class* metaklasės, kuriai taikoma ši taisyklė, pagal vardo *name* kriterijų. Taip pat nurodoma, kad surenkami elementai privalo turėti *entity* stereotipą. Taip aprašyta pirmoji taisyklės sąlyga, kuri turi kartu būti korektiška ir grąžinti teisingą *true* rezultatą tam, kad visa taisyklė būtų korektiška ir veiktų pagal aprašymą. Antroji taisyklės sąlyga taip pat naudoja *let* išraišką. Šioje taisyklės dalyje tikrinama, ar sekų diagramos gyvavimo linija yra panaudota iš klasių diagramos. Čia taip pat nurodomas laikinas kintamasis *life*, kuriame surenkamas *Type* tipo masyvas, aprašomas rinkinyje *Collection*(*Type*). Šiam masyvui priskiriami visų *Lifeline* metaklasių objektų parametro *represents* tipai. Toliau tikrinama, ar laikiname kintamajame *life* yra tikrinamasis objektas. Jeigu yra, tuomet teigiama, kad taisyklė validi, išbaigtumo diagramos esybių klasių egzemplioriai panaudoti kaip sekų diagramos gyvavimo linijos. Paveiksle 3.5 pateikiamas metaklasių modelis, kuriame

vaizduojama, kokios metaklasės buvo naudojamos kuriant trečiąją taisyklę. Paveiksle 3.6 pateiktas programos vaizdas, kaip atrodo programos langas, kai suveikia trečioji taisyklė, kai padaroma klaida kuriant projektą *MagicDraw* programoje.



3.5 pav. Trečiojoje taisyklėje naudojamos metaklasės



3.6 pav. Trečiosios taisyklės taikymo pavyzdys

4. Kiekviena operacija, įtraukta į sekų diagramos pranešimą, turi būti apibrėžta klasių diagramoje. Ši OCL kalba aprašyta taisyklė realizuota taip:

context Message inv MessageFromSequenceToClass:

not self.receiveEvent.oclIsUndefined() implies self.receiveEvent.oclIsTypeOf(MessageOccurrenceSpecification) implies (

let mess: MessageOccurrenceSpecification = self.receiveEvent.oclAsType(MessageOccurrenceSpecification)

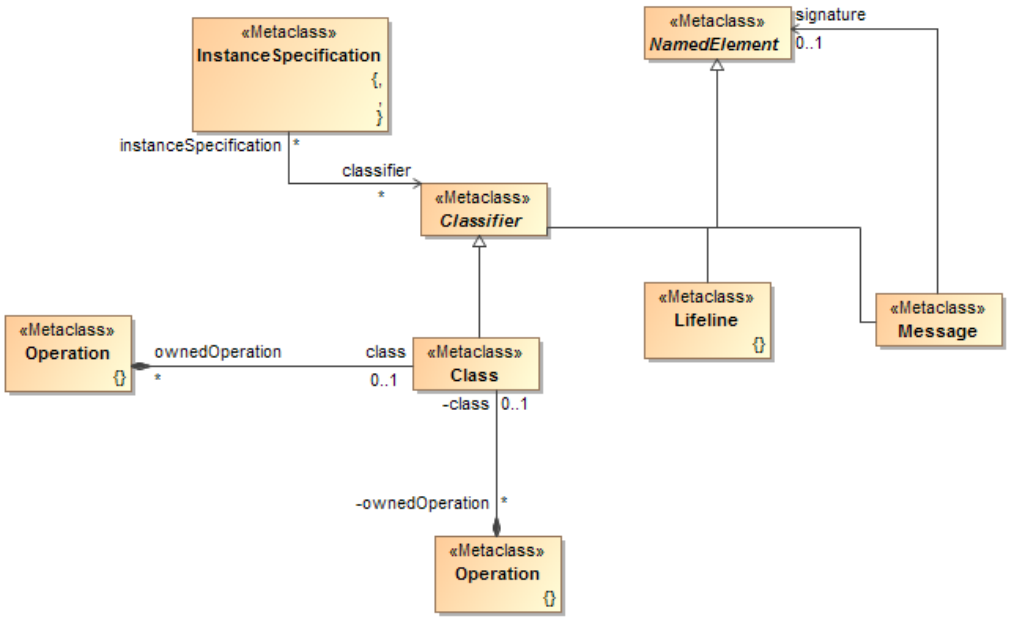
in mess.covered -> size() = 1 implies

(let life: Lifeline = mess.covered->asSequence()->first() in not life.represents.oclIsUndefined() and not life.represents.type.oclIsUndefined() and

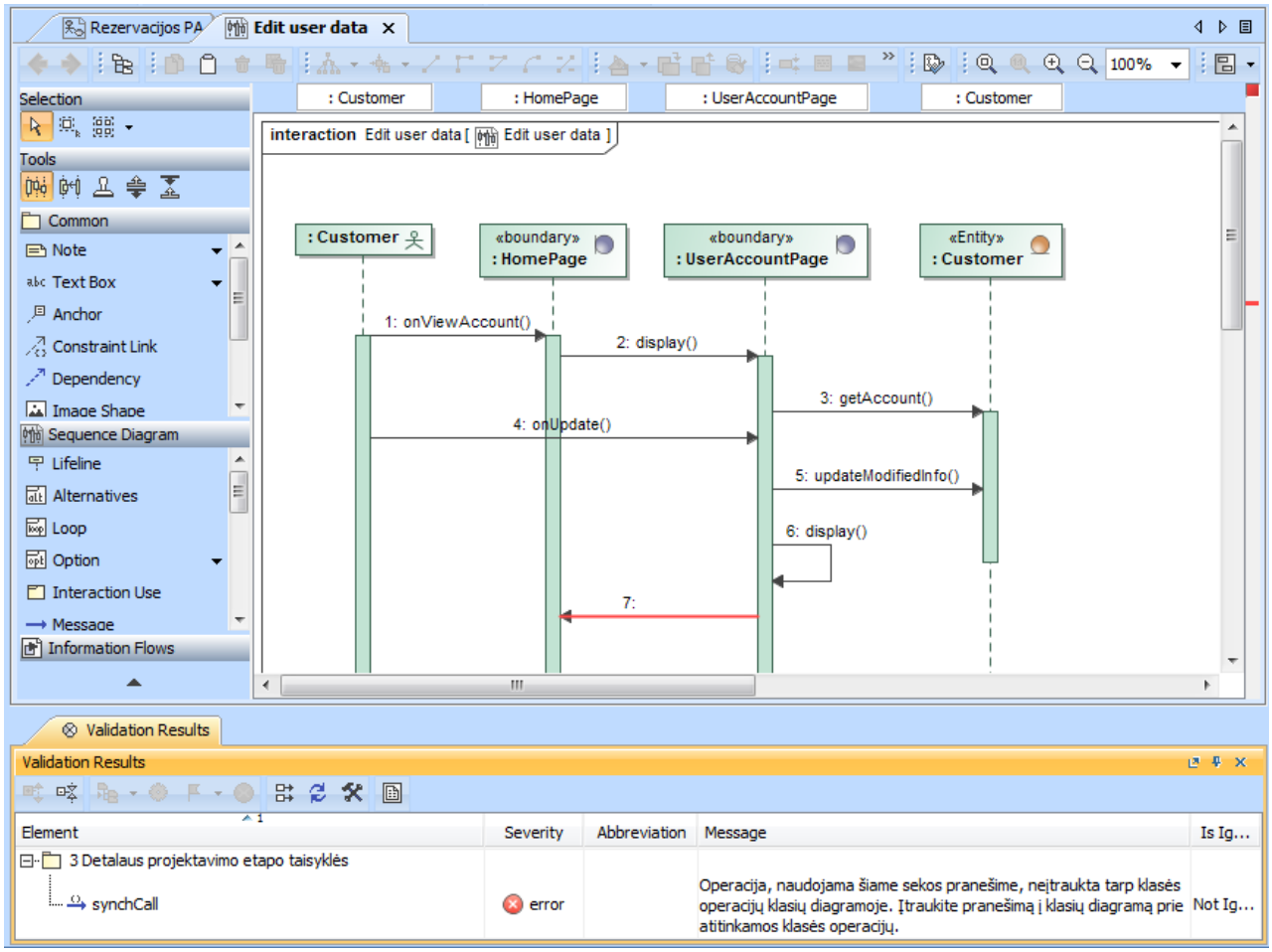
(if (life.represents.type.oclIsKindOf(Actor)) then true else

not self.signature.oclIsUndefined() and (let oper:Collection(NamedElement) = Class::allInstances()->collect(cl : Class | cl.ownedOperation) in oper->includes(self.signature)) endif)))

Ši taisyklė tikrina, ar operacija, įtraukta į sekų diagramos pranešimą, yra apibrėžta klasių diagramoje. Ši taisyklė sukurta *Message* metaklasei. Pirmiausiai patikrinama, ar parametras *receiveEvent* nėra tuščias, t. y. ar jo *self* reikšmė nėra *invalid* arba *null*. Toliau tikrinama, ar prieš tai minėtas parametras yra *MessageOccurrenceSpecification* parametro, tik tada tikrinama, ar parametro *receiveEvent* ypatybė *covered* turi lygiai vieną elementą. Jeigu taip, tuomet taisyklė vykdoma toliau. Imamas iš ankstesnių sąlygų išskaičiuotas elementas, kuris yra patalpintas masyve, ir tikrinama, ar elementas turi *represents* ypatybę, taip pat ar jame nustatyta *type* reikšmė. Jeigu rezultatas teigiamas, tada tikrinama, ar *represents.type* reikšmė yra aktorius - *Actor*. Jeigu taip, taisyklė grąžina *true* ir taisyklė toliau nebevykdoma. Jeigu reikšmė – *false*, taisyklė vykdoma toliau. Tikrinama, ar asociacijos ypatybė *signature* nustatyta. Jeigu nenustatyta, tuomet iškart taisyklė suveikia ir braukia asociaciją kaip klaidingą. Jeigu reikšmė nustatyta, tuomet imamos visos projekto klasės, surenkami klasių *ownedOperation* ir tikrinama, ar tame sąraše yra pačios asociacijos *signature* objektas. Jeigu yra, tuomet taisyklė validi, ir taisyklė negrąžina elemento kaip klaidingo. Jeigu ne, tuomet taisyklė grąžina klaidą, kad operacija neapibrėžta klasių diagramoje. Toliau paveiksle 3.7 pateikiamas metaklasių modelis, kuriame vaizduojama, kokios metaklasės buvo naudojamos kuriant ketvirtąją taisyklę. Paveiksle 3.8 pateiktas programos vaizdas, kaip atrodo programos langas, kai suveikia ketvirtoji taisyklė, kai padaroma klaida kuriant projektą *MagicDraw* programoje.



3.7 pav. Ketvirtojoje taisyklėje naudojamos metaklasės



3.8 pav. Ketvirtosios taisyklės taikymo pavyzdys

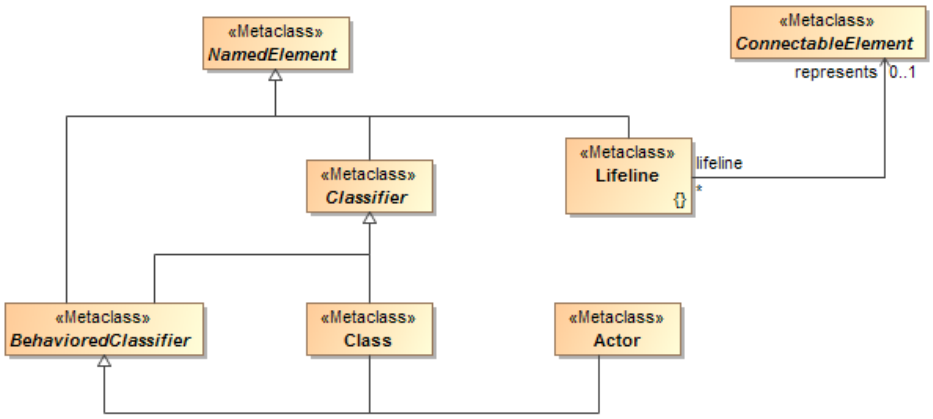
5. Kiekviena sekų diagramos gyvavimo linija (angl. *lifeline*) turi būti klasės egzempliorius (angl. *class*) klasės diagramoje. Ši OCL kalba aprašyta taisyklė realizuota taip:

context Lifeline inv LifelineIsInClassDiagram:

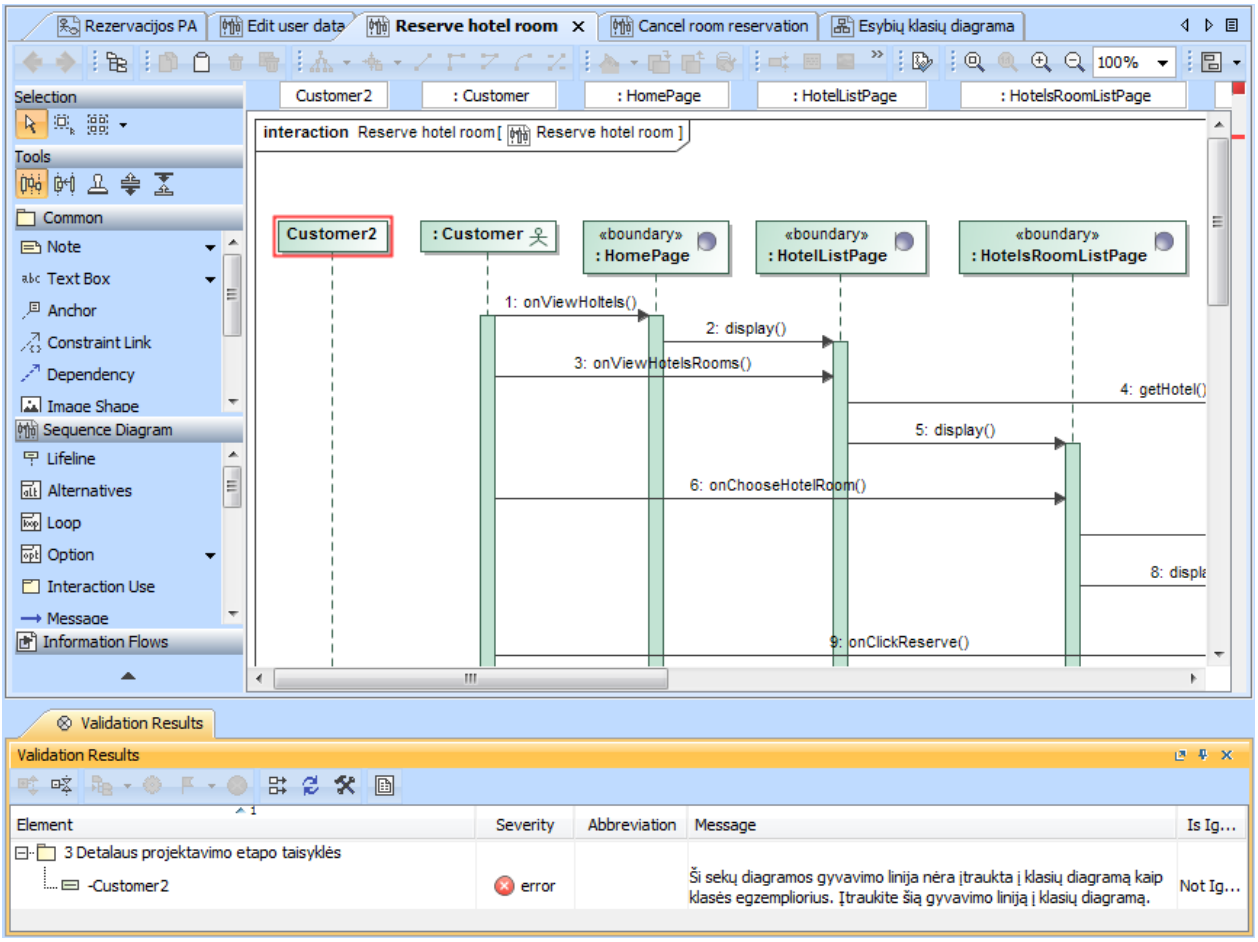
not self.represents.oclIsUndefined() and not self.represents.type.oclIsUndefined() and

(if (self.represents.type.oclIsKindOf(Actor)) then true else self.represents.type.oclIsKindOf(Class)endif)

Šioje taisyklėje tikrinama, ar sekų diagramose panaudoti gyvavimo linijos elementai jau yra apibrėžti klasės diagramoje kaip klasės egzmplioriai. Ši taisyklė sukurta *Lifeline* metaklasei. Pirmiausiai patikrinama, ar parametras *represents* nėra tuščias, t. y. ar jo *self* reikšmė nėra *invalid* arba *null*. Toliau šis pradinis patikrinimas jungiamas su kitu analogišku tikrinimu, ar parametras *represents.type* nėra tuščias, bet prieš tai patikrinama, ar tai nėra aktoriaus tipas. Jeigu tai yra aktorius, tuomet taisylė jį ignoruoja ir nelaiko klaidingu atveju. Trečioji taisyklės dalis tikrina, ar parametras *represents.type* yra klasės rūšies egzempliorius. Šitaip patikrinama, ar sekų diagramoje panaudoti gyvavimo linijų elementai buvo sukurti kaip klasės egzmplioriai klasės diagramoje. Paveiksle 3.9 pateikiamas metaklasių modelis, kuriame vaizduojama, kokios metaklasės buvo naudojamos kuriant penktąją taisyklę. Paveiksle 3.10 pateiktas programos vaizdas, kaip atrodo programos langas, kai suveikia penktoji taisyklė, kai padaroma klaida kuriant projektą *MagicDraw* programoje.



3.9 pav. Penktojoje taisyklėje naudojamos metaklasės



3.10 pav. Penktosios taisyklės taikymo pavyzdys

6. Kiekvienai klasei, kurioje operacijos naudoja kitą klasę, privalo būti priklausomybės ryšys tarp šių klasių klasių diagramoje. Ši OCL kalba aprašyta taisyklė realizuota taip:

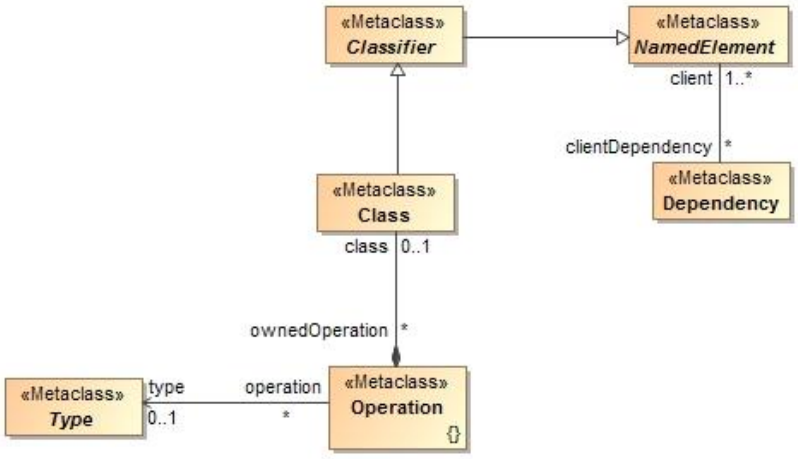
context Class inv OperationFromOtherClass:

let types:Collection(Type) = self.ownedOperation->collect(o : Operation | o.type)->select(t:Type|t.oclIsKindOf(Class)) in types->size() > 0

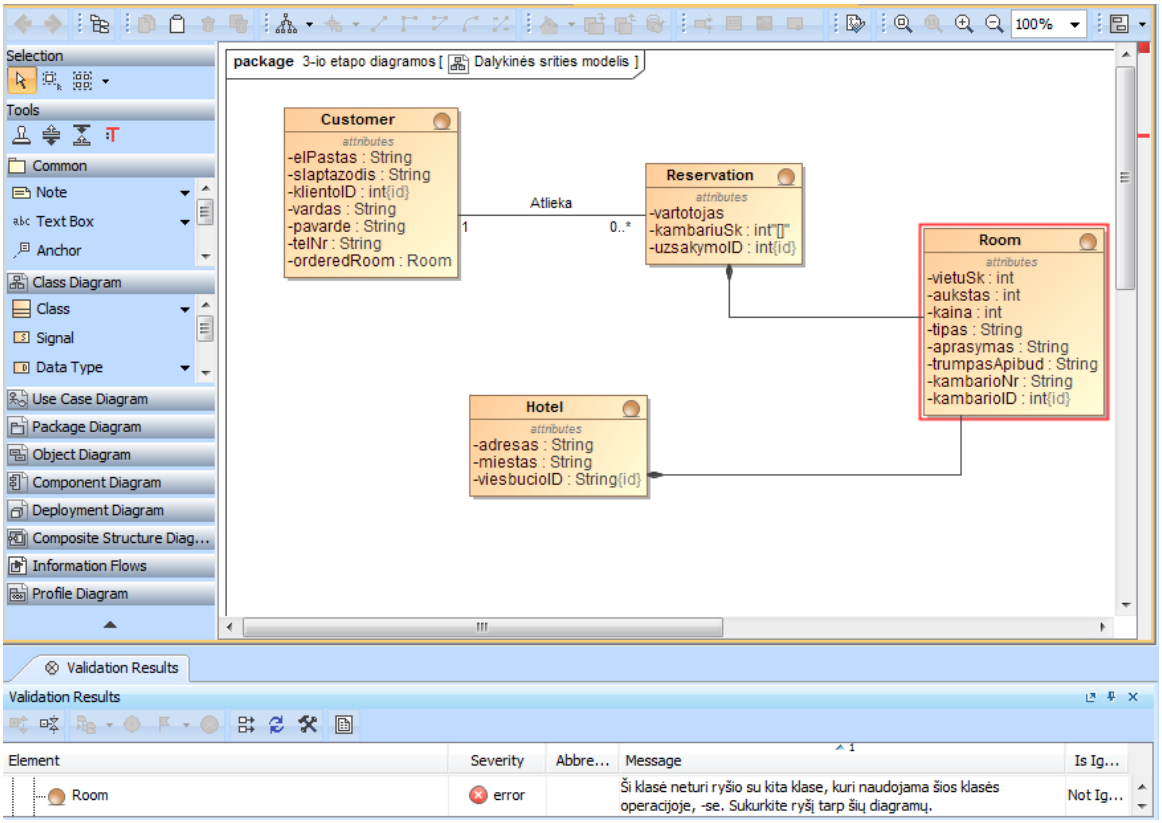
implies

(let elem:Collection(Element) = self.clientDependency->collect(dep : Dependency | dep .target)in types->forAll(t:Type | elem->includes(t.oclAsType(Element))))

Ši taisyklė tikrina, ar tarp klasių, kur viena iš jų naudoja kitą klasę savo operacijoje ar operacijose, yra nubrėžtas ryšys. Taisyklėje naudojamos dvi *let* išraiškos, jau minėtos kitose taisyklėse. Pirmoji taisyklės *let* išraiška sukuria tipų rinkinį, kuris patalpinamas po laikinu kintamuoju *types*. Į šį rinkinį surenkami elementai pagal savo operacijų tipus, o iš pačių tipų padaromas filtras, kur tipas nurodomas kaip klasė. Taip pat patikrinama, ar tarp surinktų reikšmių yra bent vienas klasės tipas. Jeigu tokių elementų nerandama, taisyklė toliau nebevykdoma, taisyklė nesuveikia nei vieno karto. Antroji taisyklės dalis taip pat naudoja *let* išraišką. Čia surenkamas elementų rinkinys pavadinimu *elem*, kurį sudaro elementai. Šie elementai yra surinkti iš *class* elemento parametro *clientDependency*, kuriame yra rinkinys elementų *Dependency*, kuriame yra parametras *target*. Galutinė sąlyga tikrina, ar kiekvienas elementas rinkinyje *types* turi savo egzempliorių rinkinyje *elem*. Taip patikrinama, ar tarp klasių yra ryšys. Paveiksle 3.11 pateikiamas metaklasių modelis, kuriame vaizduojama, kokios metaklasės buvo naudojamos kuriant šeštąją taisyklę. Paveiksle 3.12 pateiktas programos vaizdas, kaip atrodo programos langas, kai suveikia šeštoji taisyklė, kai padaroma klaida kuriant projektą *MagicDraw* programoje.



3.11 pav. Šeštojoje taisyklėje naudojamos metaklasės



3.12 pav. Šeštosios taisyklės veikimas

7. Klasės su ribiniu (angl. *boundary*) stereotipu negali turėti ryšio su kitomis ribinėmis klasėmis išbaigtumo diagramoje. Ši OCL kalba aprašyta taisyklė realizuota taip:

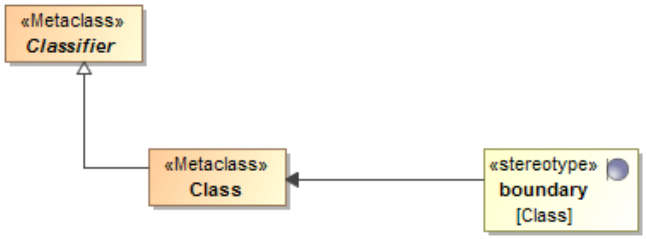
context Association inv BoundaryToBoundary:

let elements:Collection(Element) = self.relatedElement -> select(e:Element|e.appliedStereotypeInstance.oclIsUndefined() <>true) in elements->size() =2

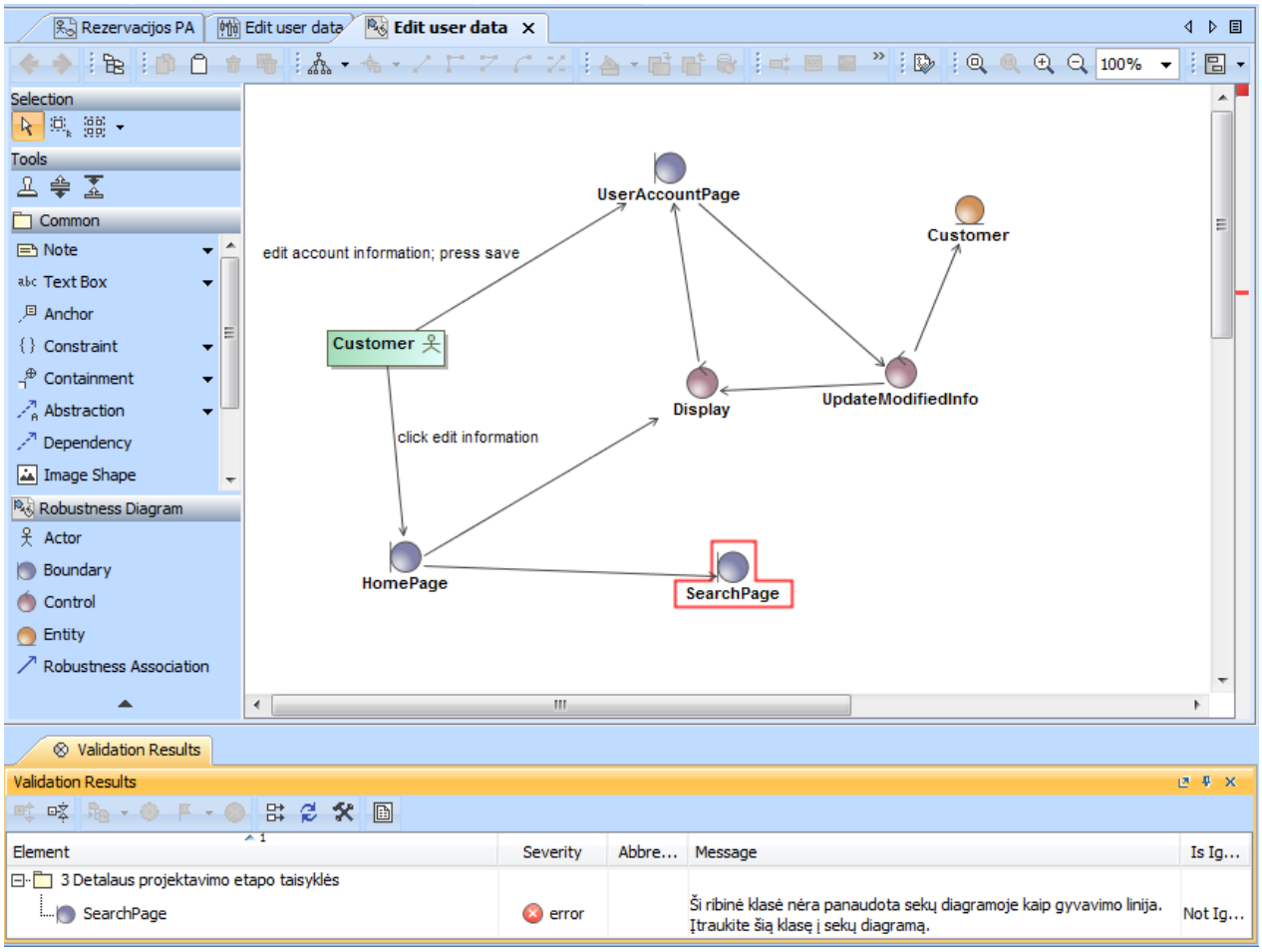
implies (

let names:Collection(String) = elements->collect(e:Element|e.appliedStereotypeInstance.classifier) -> collect(name) in not (names->count('boundary') = 2))

Ši taisyklė tikrina, ar tarp klasių, kurioms pritaikyti tokie patys ribinės klasės (angl. *boundary*) stereotipai, yra ryšys. Taisyklėje naudojamos dvi *let* išraiškos, jau minėtos kitose taisyklėse. Pirmoji taisyklės *let* išraiška sukuria elementų rinkinį, kuris patalpinamas po laikinu kintamuoju *elements*. Jame patalpintos iš asociacijos paimtos *relatedElement* lauko reikšmės. Toliau einant per kiekvieną elementą tikrinama, ar ant jo yra nustatyta *appliedStereotypeInstance* ypatybė ir jeigu jų skaičius yra du, tada surenkamos jų parametro *classifier* reikšmės naudojant išraišką *appliedStereotypeInstance.classifier – collect name.* Vėliau skaičiuojama, ar šie surinkti vardai yra ribinės klasės atitikmenys masyve. Jeigu jie yra du, tuomet taisyklė suveikia, nes tai reiškia, kad asociacijos abu galai rodo į klases su ribinės klasės stereotipais. Toliau paveiksle 3.13 pateikiamas metaklasių modelis, kuriame vaizduojama, kokios metaklasės naudojamos kuriant septintąją taisyklę. Paveiksle 3.14 pateiktas programos vaizdas, kaip atrodo programos langas, kai suveikia septintoji taisyklė, kai padaroma klaida kuriant projektą *MagicDraw* programoje.



3.13 pav. Septintojoje taisyklėje naudojamos metaklasės



3.14 pav. Septintosios taisyklės taikymo pavyzdys

8. Klasės su esybės (angl. *entity*) stereotipu negali turėti ryšio su kitomis esybių klasėmis išbaigtumo diagramoje. Ši OCL kalba aprašyta taisyklė realizuota taip:

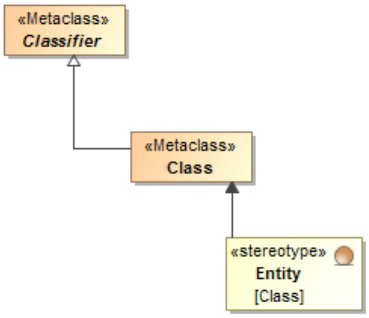
context Association inv EntityToEntity:

let elements:Collection(Element) = self.relatedElement -> select(e:Element|e.appliedStereotypeInstance.oclIsUndefined() <>true) in elements->size() =2

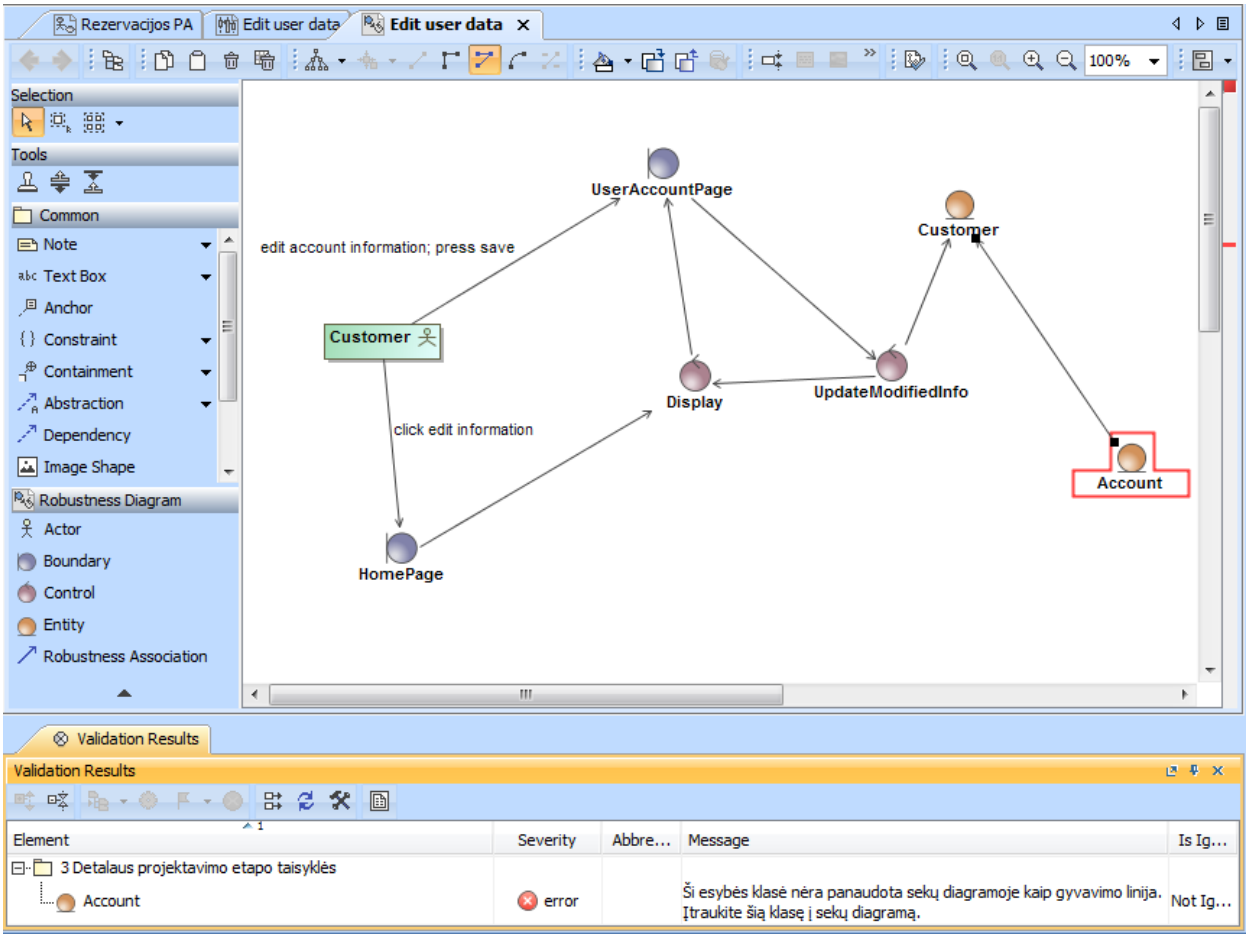
implies (

let names:Collection(String) = elements->collect(e:Element|e.appliedStereotypeInstance.classifier) -> collect(name) in not (names->count('Entity') = 2))

Ši taisyklė tikrina, ar tarp klasių, kurioms pritaikyti tokie patys esybės klasės (angl. *entity*) stereotipai, yra ryšys. Taisyklėje naudojamos dvi *let* išraiškos, jau minėtos kitose taisyklėse. Pirmoji taisyklės *let* išraiška sukuria elementų rinkinį, kuris patalpinamas po laikinu kintamuoju *elements*. Jame patalpintos iš asociacijos paimtos *relatedElement* lauko reikšmės. Toliau einant per kiekvieną elementą tikrinama, ar ant jo yra nustatyta *appliedStereotypeInstance* ypatybė ir jeigu jų skaičius yra du, tada surenkamos jų parametro *classifier* reikšmės naudojant išraišką *appliedStereotypeInstance.classifier – collect name.* Vėliau skaičiuojama, ar šie surinkti vardai yra esybės klasės atitikmenys masyve. Jeigu jie yra du, tuomet taisyklė suveikia, nes tai reiškia, kad asociacijos abu galai rodo į klases su esybės klasės stereotipais. Toliau paveiksle 3.15 pateikiamas metaklasių modelis, kuriame vaizduojama, kokios metaklasės buvo naudojamos kuriant aštuntąją taisyklę. Paveiksle 3.16 pateiktas programos vaizdas, kaip atrodo programos langas, kai suveikia aštuntoji taisyklė, kai padaroma klaida kuriant projektą *MagicDraw* programoje.



3.15 pav. Aštuntojoje taisyklėje naudojamos metaklasės



3.16 pav. Aštuntosios taisyklės taikymo pavyzdys

9. Klasės su ribiniu (angl. *boundary*) stereotipu negali turėti ryšio su kitomis esybių (angl. *entity*) klasėmis išbaigtumo diagramoje. Ši OCL kalba aprašyta taisyklė realizuota taip:

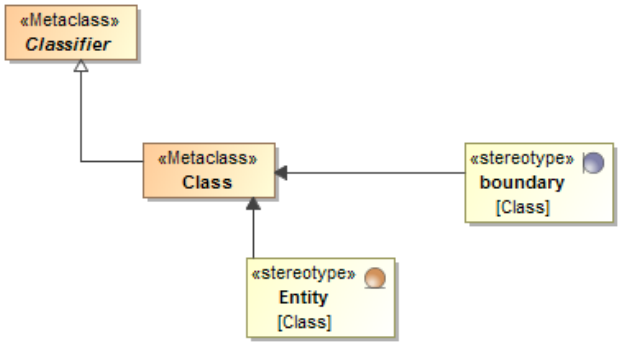
context Association inv BoundaryToEntity:

let elements:Collection(Element) = self.relatedElement -> select(e:Element|e.appliedStereotypeInstance.oclIsUndefined() <>true) in elements->size() =2

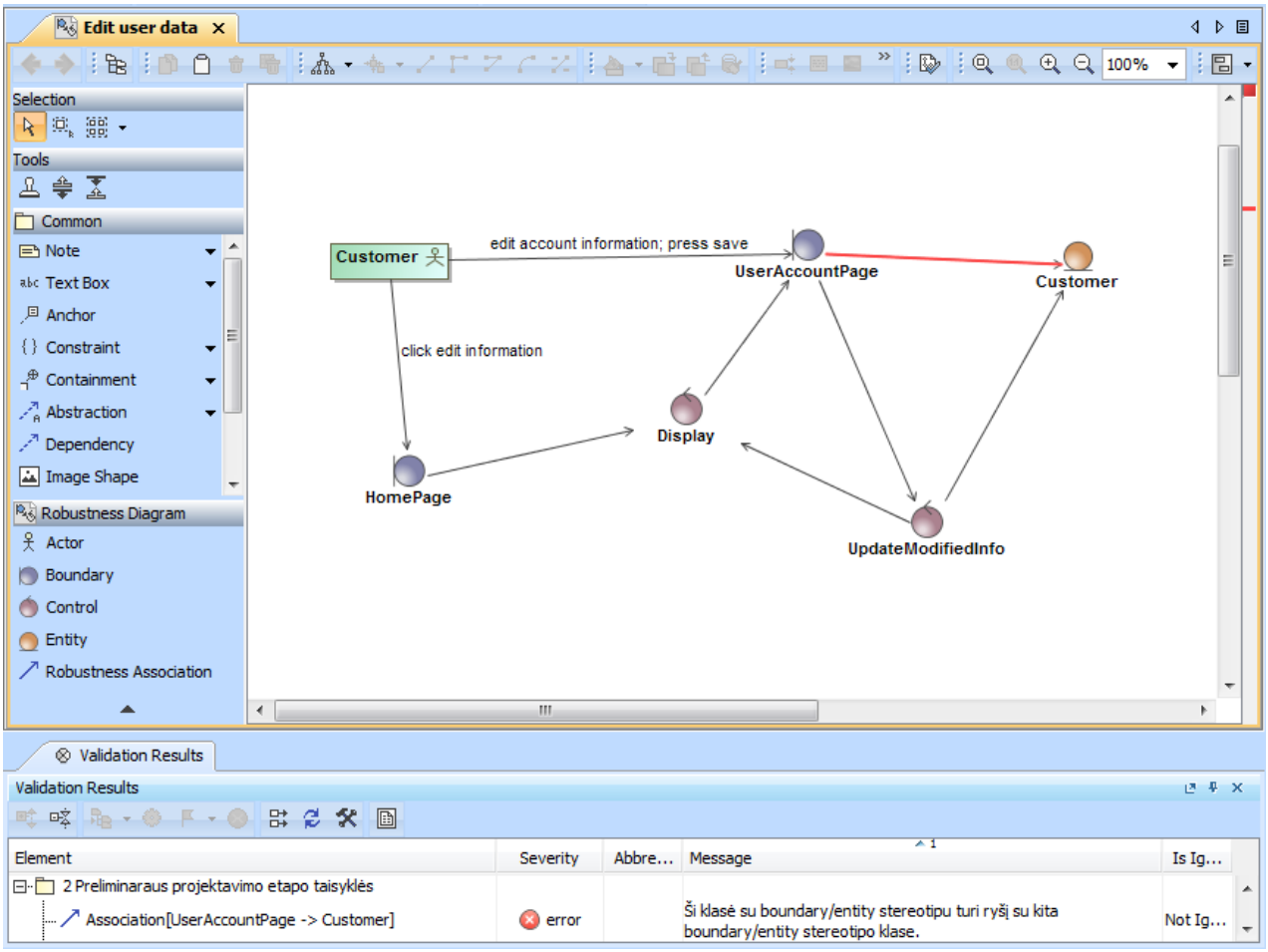
implies (

let names:Collection(String) = elements->collect(e:Element|e.appliedStereotypeInstance.classifier) -> collect(name) in not (names->includes('boundary') and names->includes('Entity')))

Ši taisyklė tikrina, ar tarp klasių, kurioms pritaikyti ribinės (angl. *boundary*) arba esybės klasės (angl. *entity*) stereotipai, yra ryšys. Taisyklėje naudojamos dvi *let* išraiškos, jau minėtos kitose taisyklėse. Pirmoji taisyklės *let* išraiška sukuria elementų rinkinį, kuris patalpinamas po laikinu kintamuoju *elements*. Jame patalpintos iš asociacijos paimtos *relatedElement* lauko reikšmės. Toliau einant per kiekvieną elementą tikrinama, ar ant jo yra nustatyta *appliedStereotypeInstance* ypatybė ir jeigu jų skaičius yra du, tada surenkamos jų parametro *classifier* reikšmės naudojant išraišką *appliedStereotypeInstance.classifier – collect name.* Vėliau skaičiuojama, ar šie surinkti vardai yra vienas iš jų ribinės klasės atitikmuo, o kitas – esybės minėtame masyve. Jeigu nustatomi abu stereotipai, tuomet taisyklė suveikia, nes tai reiškia, kad asociacijos abu galai rodo į klases su ribinės arba esybės klasės stereotipais. Toliau paveiksle 3.17 pateikiamas metaklasių modelis, kuriame vaizduojama, kokios metaklasės buvo naudojamos kuriant devintąją taisyklę. Paveiksle 3.18 pateiktas programos vaizdas, kaip atrodo programos langas, kai suveikia devintoji taisyklė, kai padaroma klaida kuriant projektą *MagicDraw* programoje.



3.17 pav. Devintojoje taisyklėje naudojamos metaklasės



3.18 pav. Devintosios taisyklės taikymo pavyzdys

10. Aktoriaus klasės negali turėti ryšio su esybės (angl. *entity*) stereotipo klasėmis išbaigtumo diagramoje. Ši OCL kalba aprašyta taisyklė realizuota taip:

context Association inv ActorToEntity:

self.relatedElement->size() = 2 implies (

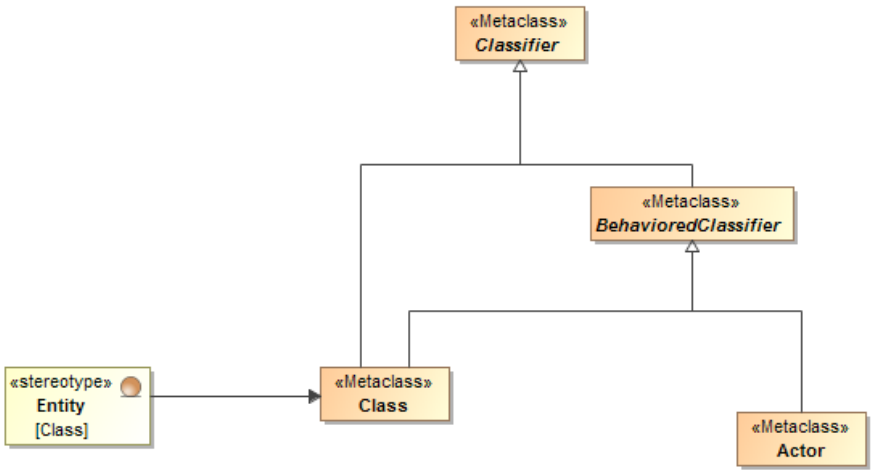
let actors:Collection(Element) = self.relatedElement -> select(e:Element|e.oclIsTypeOf(Actor)),

elements:Collection(Element) = self.relatedElement -> select(e:Element|e.appliedStereotypeInstance.oclIsUndefined() <>true)

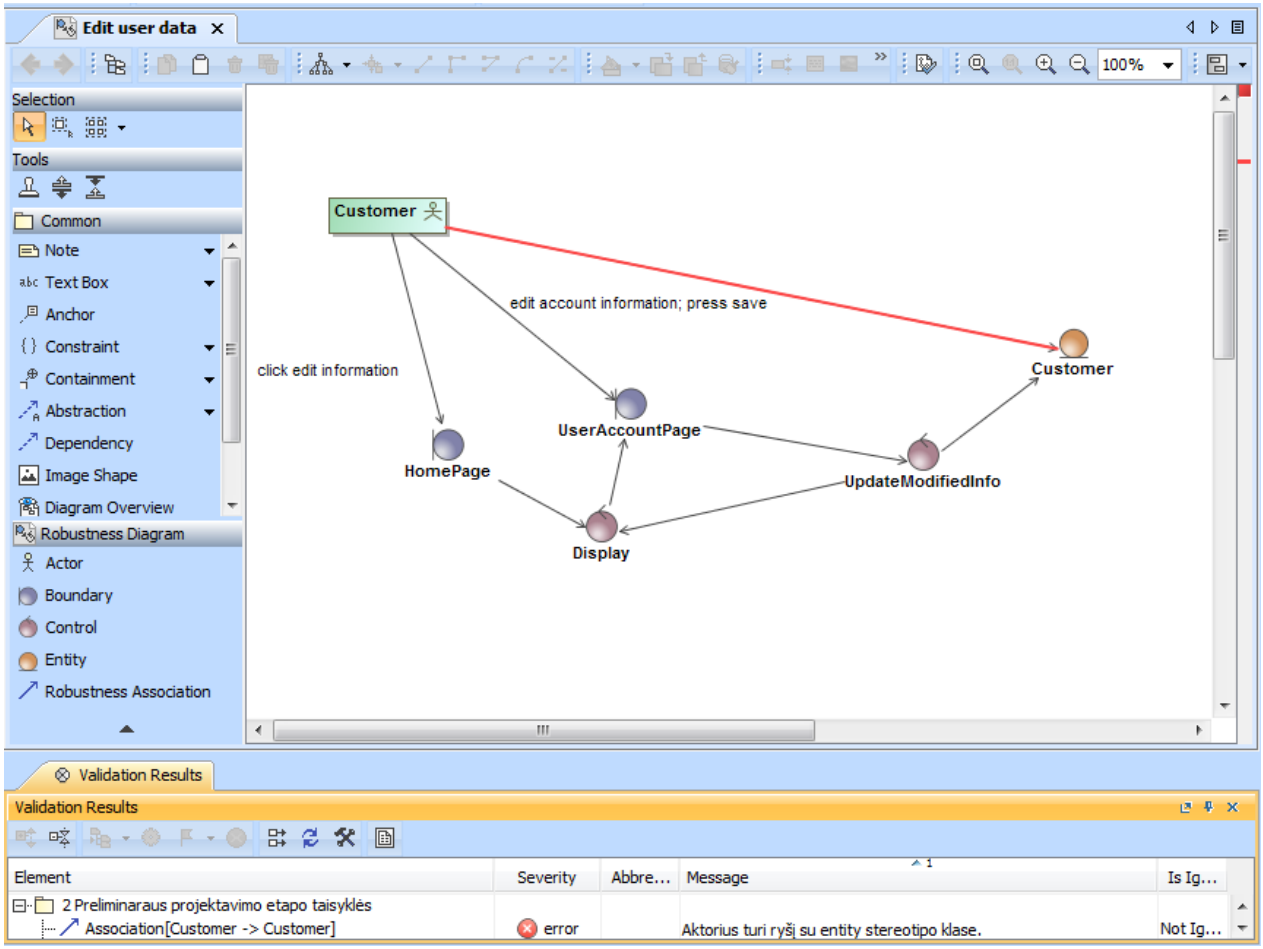
in (actors->size() = 1 and elements->size() =1) implies (

let names:Collection(String) = elements->collect(e:Element|e.appliedStereotypeInstance.classifier) -> collect(name) in not (names->includes('Entity'))))

Šia taisykle tikrinama, ar aktorius turi turi ryšį su esybės stereotipo klase. Iš pradžių patikrinama, ar *relatedElement* parametrai yra du, ir jeigu taip - tikrinama toliau. Tada tikrinama, ar elementas yra aktorius *Actor*, jeigu taip – elementas saugomas į *actors* masyvą. Tada surenkami visi *relatedElement* stereotipo elementai. Tada tikrinama, ar tarp *actors* masyve esančių elementų yra nors vienas aktorius ir ar *elements* masyvas turi vieną elementą. Tada tikrinama ar elementų masyve yra nors vienas esybės stereotipas. Jeigu taip, tada taisyklė suveikia, kad negali aktorius turėti ryšio su esybės stereotipo klase. Toliau paveiksle 3.19 pateikiamas metaklasių modelis, kuriame vaizduojama, kokios metaklasės buvo naudojamos kuriant dešimtąją taisyklę. Paveiksle 3.20 pateiktas programos vaizdas, kaip atrodo programos langas, kai suveikia dešimtoji taisyklė, kai padaroma klaida kuriant projektą *MagicDraw* programoje.



3.19 pav. Dešimtojoje taisyklėje naudojamos metaklasės



3.20 pav. Dešimtosios taisyklės taikymo pavyzdys

11. Aktoriaus klasės negali turėti ryšio su valdiklio (angl. *control*) stereotipo klasėmis išbaigtumo diagramoje. Ši OCL kalba aprašyta taisyklė realizuota taip:

context Association inv ActorToControl:

self.relatedElement->size() = 2 implies (

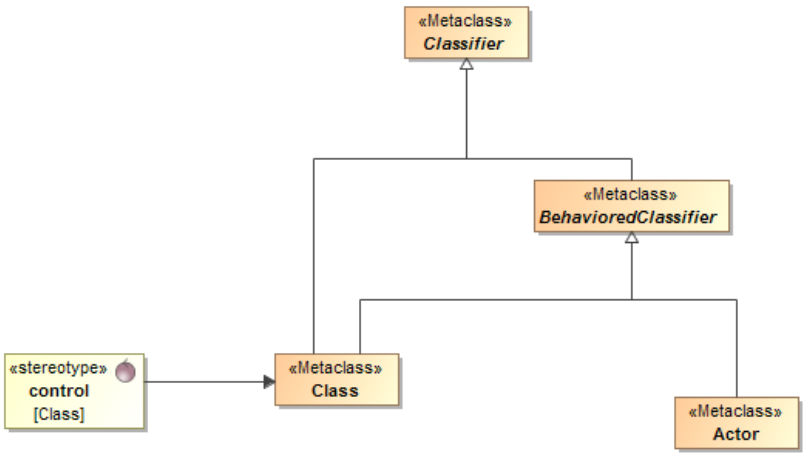
let actors:Collection(Element) = self.relatedElement -> select(e:Element|e.oclIsTypeOf(Actor)),

elements:Collection(Element) = self.relatedElement -> select(e:Element|e.appliedStereotypeInstance.oclIsUndefined() <>true)

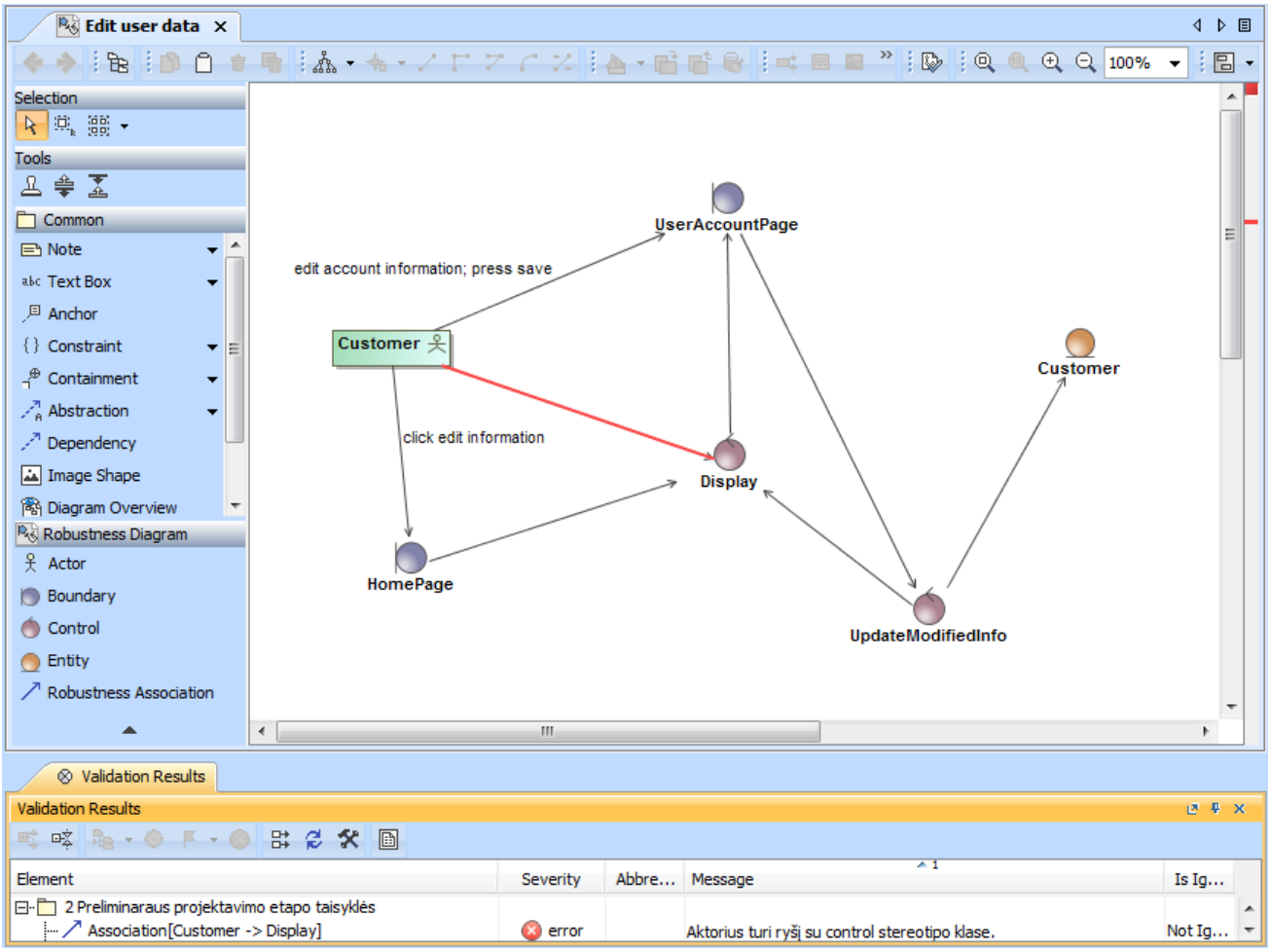
in (actors->size() = 1 and elements->size() =1) implies (

let names:Collection(String) = elements->collect(e:Element|e.appliedStereotypeInstance.classifier) -> collect(name) in not (names->includes('control'))))

Šia taisykle tikrinama, ar aktorius turi ryšį su valdiklio stereotipo klase. Iš pradžių patikrinama, ar *relatedElement* parametrai yra du, ir jeigu taip - tikrinama toliau. Tada tikrinama, ar elementas yra aktorius *Actor*, jeigu taip – elementas saugomas į *actors* masyvą. Tada surenkami visi *relatedElement* stereotipo elementai. Tada tikrinama, ar tarp *actors* masyve esančių elementų yra nors vienas aktorius ir ar *elements* masyvas turi vieną elementą. Tada tikrinama ar elementų masyve yra nors vienas valdiklio stereotipas. Jeigu taip, tada taisyklė suveikia, kad negali aktorius turėti ryšio su valdiklio stereotipo klase. Toliau paveiksle 3.21 pateikiamas metaklasių modelis, kuriame vaizduojama, kokios metaklasės buvo naudojamos kuriant vienuoliktąją taisyklę. Paveiksle 3.22 pateiktas programos vaizdas, kaip atrodo programos langas, kai suveikia vienuoliktoji taisyklė, kai padaroma klaida kuriant projektą *MagicDraw* programoje.



3.21 pav. Vienuoliktojoje taisyklėje naudojamos metaklasės



3.22 pav. Vienuoliktosios taisyklės taikymo pavyzdys

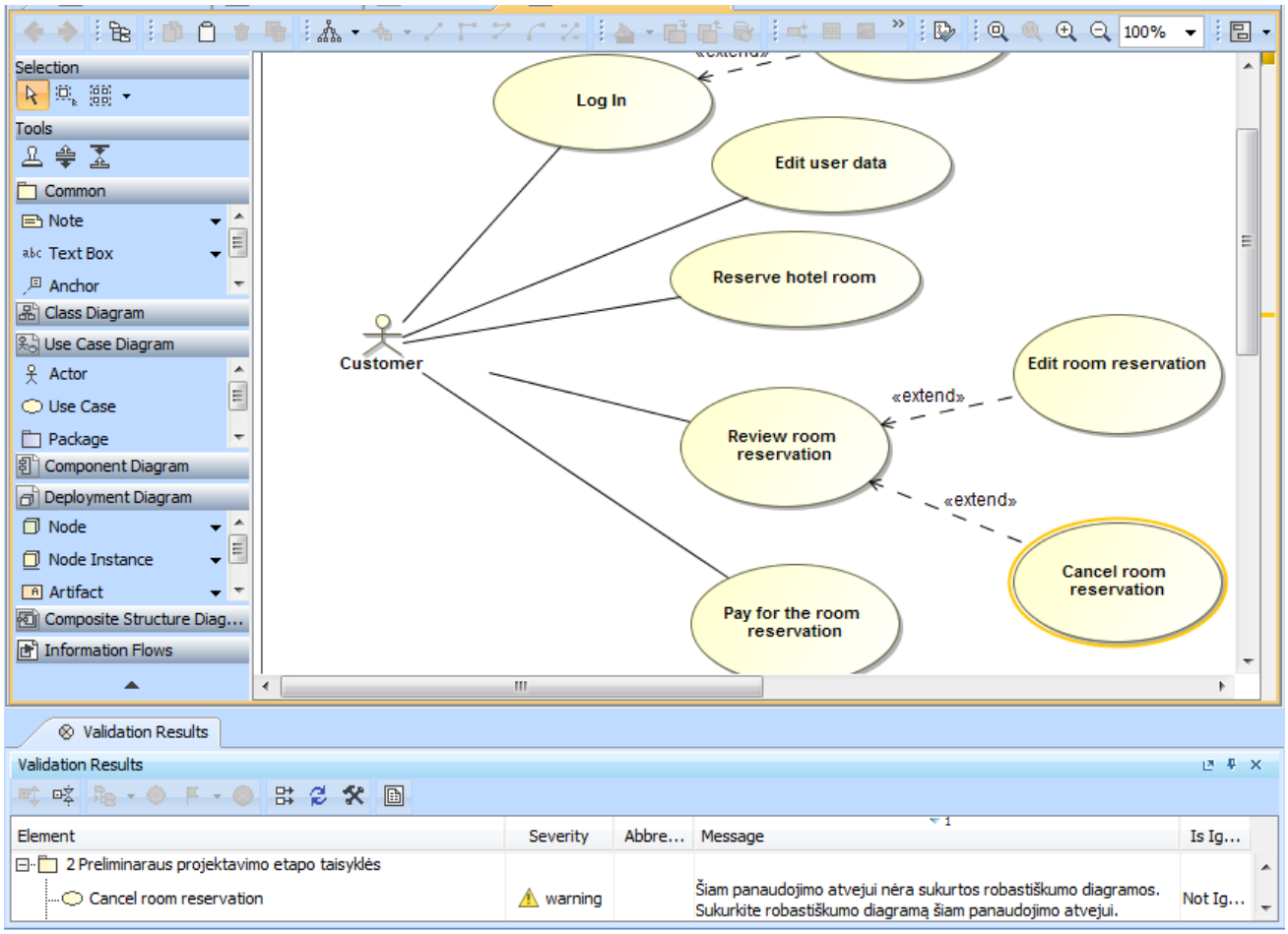
12. Kiekvienam panaudojimo atvejui sudaroma detalizuota išbaigtumo (angl. *robustness*) diagrama.

Ši taisyklė kitaip negu kitos yra parašyta *Java* kalba. Šios taisyklės išreikšti OCL išraiška nepavyko, nes taisyklių kūrimo metu dar nebuvo išleista naujausia *MagicDraw* programos versija, kurioje jau galima tikrinti ne tik diagramų elementus, bet ir pačias diagramas naudojant OCL kalbą. *Java* kalba aprašyta taisyklė pateikiama žemiau.

```
public static boolean isValid(UseCase useCase) {
        Collection<Diagram> diagrams = useCase.getOwnedDiagram();
        if (diagrams.isEmpty()) return false;
        for (Diagram diagram : diagrams) {
                if (diagram.get_representation().getType().equals("Robustness Diagram")) return true;
                }
                return false;     }}
```

Ši taisyklė tikrina, ar panaudojimo atvejui sukurta išbaigtumo diagrama. Iš pradžių diagramos surenkamos į diagramų rinkinį, kuriame patalpinamos visos panaudojimo atvejams sukurtos diagramos. Surinkus visas diagramas į rinkinį, pirmiausiai patikrinama, ar išvis panaudojimo atvejui yra sukurta diagrama. Jeigu nėra sukurta nei vienos diagramos, iškart grąžinama klaida, kad nėra sukurtos diagramos. Jeigu aptinkama bent viena diagrama, sukurta panaudojimo atvejui, tada imama po vieną diagramą ir tikrinama, ar diagramos tipas yra išbaigtumo diagrama. Jeigu reikšmė teigiama, tuomet taisyklė nerodo elemento kaip klaidingo. Jeigu reikšmė neigiama, t. y. diagramos tipas nėra išbaigtumo diagrama, tada grąžinama klaida. Suveikusi taisyklė informuoja,

kad panaudojimo atvejui nėra sukurtos išbaigtumo diagramos. Suveikusios taisyklės *MagicDraw* programoje ekrano vaizdas pateikiamas 3.23 paveiksle.



3.23 pav. Dvyliktosios taisyklės taikymo pavyzdys

Visos aprašytos taisyklės taip pat turi ir kiekvienai iš jų nustatytą klaidos įspėjimo lygį. Taisyklės klaidos rimtumo lygis nustatytas pagal tai, ar taisyklė pritaikyta tik *ICONIX* metodui, ar ne. Jeigu taisyklė taikoma tik kaip *ICONIX* metode taikoma, tuomet jos klaidos rimtumo lygis žymimas kaip klaida (angl. *error*). Visos likusios taisyklės, kurios aktualios *ICONIX* metodu kuriamam projektui, tačiau taisyklės nėra tokios svarbios norint užtikrinti *ICONIX* metodu kurto projekto elementų suderinamumui, tada klaidos lygis žymimas kaip įspėjimas (angl. *warning*). Abejais atvejais vartotojas, atlikęs projekto validaciją, gali pats nuspręsti, ar jis nori taisyti validacijos įrankio aptiktas klaidas, įspėjimus, ar jam tai neaktualu, ir projektas yra korektiškas. Taisyklės priskyrimas tam tikram klaidos lygiui pateiktas 3-ioje lentelėje.

3 lentelė. Taisyklių įspėjimo lygio priskyrimas taisyklei

Nr.	Taisyklės nr.	Etapas	Taisyklė	Trumpas taisyklės pavadinimas	Įspėjimo lygis
1.	2.1.	II	Kiekvienam panaudojimo atvejui sudaroma detalizuota išbaigtumo diagrama.	*UseCaseHasRobustness*	Klaida (angl. error)
2.	2.2.	II	Klasės su ribiniu (angl. boundary) stereotipu negali turėti ryšio su kitomis ribinėmis (angl. boundary) klasėmis išbaigtumo diagramoje.	*BoundaryToBoundary*	Klaida (angl. error)
3.	2.3.	II	Klasės su ribiniu (angl. boundary) stereotipu negali turėti ryšio su esybės (angl. entity) klasėmis išbaigtumo diagramoje.	*BoundaryToEntity*	Klaida (angl. error)
4.	2.4.	II	Klasės su esybės (angl. entity) stereotipu negali turėti ryšio su esybės (angl. entity) klasėmis išbaigtumo diagramoje.	*EntityToEntity*	Klaida (angl. error)
5.	2.5.	II	Aktoriai negali turėti ryšio su esybės (angl. entity) stereotipo klasėmis išbaigtumo diagramoje.	*ActorToEntity*	Klaida (angl. error)
6.	2.6.	II	Aktoriai negali turėti ryšio su valdiklio (angl. control) stereotipo klasėmis išbaigtumo diagramoje.	*ActorToControl*	Klaida (angl. error)
7.	3.1.	III	Kiekviena operacija, įtraukta į sekų diagramos pranešimą, turi būti apibrėžta klasių diagramoje.	*MessageFromSequenceToClass*	Įspėjimas (angl. warning)
8.	3.2.	III	Kiekviena sekų diagramos gyvavimo linija (angl. lifeline) turi būti klasės egzempliorius (angl. class) klasės diagramoje.	*LifelineIsInClassDiagram*	Įspėjimas (angl. warning)
9.	3.3.	III	Kiekvienai klasei, kurioje operacijos naudoja kitą klasę, privalo būti priklausomybės ryšys tarp šių klasių klasių diagramoje.	*OperationFromOtherClass*	Įspėjimas (angl. warning)
10.	3.4.	III	Kiekvienam panaudojimo atvejui sudaroma detalizuota sekų diagrama.	*UsecaseHasSequence*	Klaida (angl. error)
11.	3.5.	III	Kiekvienas išbaigtumo diagramos ribinės (angl. boundary) klasės egzempliorius turi būti panaudotas kaip sekų diagramos gyvavimo linija (angl. lifeline).	*BoundaryAsLifeline*	Klaida (angl. error)
12.	3.6.	III	Kiekvienas išbaigtumo diagramos esybės (angl. entity) klasės egzempliorius turi būti panaudotas kaip sekų diagramos gyvavimo linija (angl. lifeline).	*EntityAsLifeline*	Klaida (angl. error)

3.2. Rekomendacijos kuriant projektą

Šiuose nurodymuose trumpai pateikiamos rekomendacijos, kokius žingsnius reikia atlikti, norint lengviau atlikti validaciją modeliams, sukurtiems pagal ICONIX metodiką *MagicDraw* aplinkoje. Grafinis vaizdas, kokie pagrindiniai žingsniai rekomenduojami atlikti, pateikti pav. nr. 3.24.

1. Reikalavimų etapas

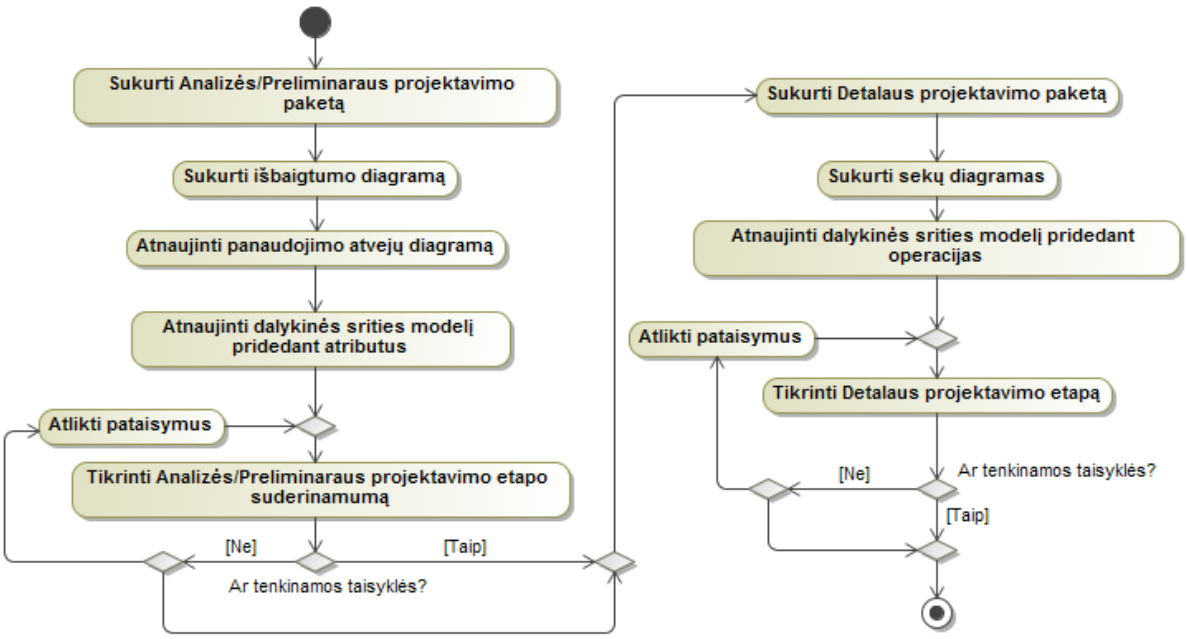
MagicDraw sukurtame projekte sukurkite paketą, kuriame talpinsite reikalavimų etapo diagramas. Apsibrėžkite, kokie bus sistemos funkciniai reikalavimai. Sukurkite pradinį dalykinės srities modelį sukurtame reikalavimų etapo pakete. Nuspręskite, kaip vartotojas sąveikaus su sistema – sukurkite pirmąją panaudojimo atvejų diagramą. Rekomenduojama identifikuoti bent pagrindinius panaudojimo atvejus, kurie bus realizuoti. Įsitikinkite, kad panaudojimo atvejai atitinka užsakovo reikalavimus.

2. Preliminaraus projektavimo etapas

Sukurkite dar vieną paketą, kuriame talpinsite preliminaraus projektavimo diagramas. Išbaigtumo diagramas kurkite atskirame aplanke nuo likusių šio etapo diagramų. Sukurkite būtinai išbaigtumo tipo (angl. *robustness*) diagramą (ne klasių) perpanaudojant panaudojimo atvejuose aprašytą informaciją. Kol kuriate išbaigtumo diagramas, atnaujinkite dalykinės srities modelį. Pridėkite trūkstamas klases, ištaisykite netikslumus, taip pat dalykinės srities objektams pridėkite atributus. Perkelkite panaudojimo atvejus. Tam, kad nereikėtų diagramų kurti iš naujo, panaudokite *MagicDraw* funkciją *Model Transformations*, esančią *Tools* skiltyje. Pasinaudoję šia funkcija lengvai klonuosite jau sukurtas diagramas į kitą pasirinktą projekto vietą. Galėsite toliau dirbti su jau sukurtomis diagramomis jas papildant, bet taip pat išsaugant ir pradinę diagramos versiją. Sukūrus naujas ir atnaujinus jau sukurtas diagramas preliminaraus projektavimo etape, atlikite antrojo etapo diagramų validaciją. Programoje rinkitės *Analyze -> Validation -> Validate*. Atsidariusiame lange pasirinkite preliminaraus projektavimo etapo validacijos taisyklių paketą, nurodykite aplanką, kuriame sukūrėte diagramas, spauskite *Validate*. Programa patikrins sukurtas reikalavimų etapo diagramas pagal ICONIX metodui pritaikytas taisykles. Jeigu bus klaidų – programa informuos, klaidas bus galima pataisyti arba jų nepaisyti.

3. Detalaus projektavimo etapas

Sukurkite trečią paketą, kuriame kursite trečiojo etapo diagramas. Pakete sukurkite sekų diagramas, kuriose detaliai aprašysite, kaip realizuosite panaudojimo atvejus. Taip pat atnaujinkite dalykinės srities modelį, dalykinės srities objektams pridėkite operacijas. Kaip iš vieno paketo į kitą persikelti jau sukurtas diagramas aparašėme antrame etape. Sukūrus ir atnaujinus diagramas, atlikite trečiojo etapo diagramų validaciją. Atlikite analogiškus žingsnius kaip ir antrame etape, tik rinkitės trečiojo etapo taisykles ir kątik sukurtų diagramų paketą. Atlikus validaciją, matysite validacijos rezultatus aptiktų klaidų skirtuke arba sėkmingai atliktos validacijos pranešimą.



3.24 pav. Rekomendacijų kuriant projektą grafinė iliustracija

Laikantis pateiktų rekomendacijų kuriamam modeliui, validacijos taisyklės gali būti lengviau pritaikomos. Šių rekomendacijų galima ir nesilaikyti, tačiau tada didesnė tikimybė, kad norint pritaikyti validacijos taisykles reikės atlikti daugiau paruošiamųjų žingsnių, kol bus galima pritaikyti validacijos taisykles ir šios pateiks korektiškus rezultatus. Verta paminėti, kad taisykles taikant ne numatytam sukurtų diagramų etapui, o visiškai kitam, arba tiesiog visam modeliui, rezultatai bus gauti netikslūs, taisyklės tikėtina suveiks ne tose modelio vietose, kuriose buvo numatyta.

4. EKSPERIMENTINIS TAISYKLIŲ RINKINIO TYRIMAS

4.1. Eksperimento planas

Eksperimento tikslas – patikrinti sukurtų validacijos taisyklių veikimą *MagicDraw* projektuose, kurtuose pagal ICONIX metodą. Eksperimentą sudaro trys dalys:

1. Pirmoje eksperimento dalyje sukurtas projektas, aprašytas D.Rosenberg‘o ir K.Scott‘o knygoje „Applying use case driven object modeling with UML“ [25]. Knygos autoriai pateikia praktinių pavyzdžių, kaip kuriamas projektas, naudojant ICONIX metodą. Aprašyti pavyzdžiai perkelti į *MagicDraw* programos aplinką ir atlikta projekto validacija naudojant sukurtas validacijos taisykles. Daugiau apie šią eksperimento dalį pateikiama kitame skyriuje.
2. Antroje eksperimento dalyje sukurtas eksperimentinis *MagicDraw* projektas, kuriame modeliuojama viešbučio kambario užsakymo ir apmokėjimo informacinė sistema. Projektas kurtas taip pat remiantis ICONIX metodu. Sukurtam projektui taip pat pritaikytos validacijos taisyklės. Gauti rezultatai pateikti kitame skyriuje.
3. Trečioje eksperimento dalyje taisyklės pritaikytos studentų atliktiems darbams. Projektai kurti viename iš universiteto modulių, kuriame dėstoma, kaip kurti informacines sistemas taikant ICONIX metodą. Surinkti 3 skirtingi studentų *MagicDraw* projektai, kuriems buvo pritaikytos validacijos taisyklės.

Visų trijų eksperimento etapų tikslas – patikrinti taisyklių veiksmingumą. Visuose etapuose naudojamos tos pačios taisyklės. Taisyklės išeksportuotos kaip *MagicDraw* profilis, kurį galima įsikelti į bet kurį *MagicDraw* projektą, ir naudoti įprastai kaip jau numatytosios *MagicDraw* programos validacijos taisyklės.

Taisyklės suskirstytos etapais – kuriame etape kurios taisyklės turėtų būti taikomos. Atskiro etapo taisyklės pateikiamos atskirame aplanke. Kaip jau minėta rekomendacijose – kuriant projektą rekomenduojama šį skirstyti taip pat etapais pagal ICONIX metode numatomus informacinės sistemos kūrimo etapus – Reikalavimų, Preliminaraus projektavimo, Detalaus projektavimo. Visose eksperimento dalyse projektai kurti (arba suskirstyti) į numatytus etapus vadovaujantis rekomendacijomis, kaip kurti projektą. Kiekvienai iš eksperimento dalių pritaikomos taisyklės, sekama, kiek ir kokių klaidų gaunama, kokios taisyklės suveikia. Taip pat pateikiama kiekvieno projekto statistika – kiek elementų sudaro projektą, kokie elementai naudojami, taip pat kiek ir kokių diagramų sukurta.

Visose eksperimento dalyse naudojamos anksčiau aprašytos taisyklės, toliau naudojama šių taisyklių numeracija pateikiama anksčiau pateiktoje 4-oje lentelėje.

4.1.1. Knygų parduotuvės IS projektas

Šis projektas kurtas remiantis ICONIX knygoje pateiktu pavyzdžiu. D.Rosenberg‘as ir K.Scott‘as knygoje „Applying use case driven object modeling with UML“ pateikia informacinės sistemos kūrimo pavyzdį taikant ICONIX metodą. Kuriamos informacinės sistemos tema – Internetinė knygų parduotuvė (angl. *Internet Bookstore*). Knyga parašyta remiantis nuosekliai kuriamos informacinės sistemos kūrimo etapais, t. y. pirmiausiai pradedama nuo reikalavimų etapo, tęsiama preliminaraus projektavimo ir t.t. Taigi knygos autoriai nuosekliai aprašo kiekvieną informacinės sistemos kūrimo etapą tiek žodžiais, tiek iliustruoja aprašus diagramų pavyzdžiais. Kaip rašo knygos autoriai, knygoje „aprašomi panaudojimo atvejai ir klasės, reikalingi išpildyti būtiniausiems reikalavimams, kuriuos klientas pateikė informacinei sistemai“. Buvo pateikti tokie pagrindinai reikalavimai:

1. Internetinėje parduotuvėje užsakymus galima pateikti internetu.
2. Internetinėje parduotuvėje vartotojų paskyros privalo būti apsaugotos slaptažodžiais.
3. Internetinėje parduotuvėje knygų galima ieškoti per paieškos funkciją.
4. Internetinė parduotuvė turi užtikrinti galimybę už knygą atsiskaityti kreditine kortele.
5. Internetinėje parduotuvėje bet kas gali pateikti knygos atsiliepimus.
6. Internetinėje parduotuvėje galima knygas reitinguoti priklausomai nuo klientų įvertinimų.

Remiantis tokiais reikalavimais, knygos autoriai aprašo kuriamą sistemą. Jų pateikti diagramų pavyzdžiai buvo perkelti į *MagicDraw* programos aplinką. Sukurtam projektui pritaikytos validacijos taisyklės norint patikrinti, ar knygoje pateikti diagramų modeliai yra korektiški ir atitinka autorių pateiktas taisykles, kurios būdingos tik ICONIX metodu kuriamiems projektams. Detalesnė knygoje pateikto projekto diagramų ir elementų sudėtis pateikiama lentelėje nr. 4.

4 lentelė. Knygų parduotuvės IS projektą sudarantys elementai

Kriterijus / Objektai		I etapo objektai	II etapo objektai	III etapo objektai
Diagramos	Klasių	1	1	1
	Panaudojimo atvejų	1	1	1
	Išbaigtumo		5	5
	Sekų			5
Elementai	Asociacijos	15	68	92
	Klasės (entity)	8	8	8
	Panaudojimo atvejai	5	5	5
	Aktoriai	4	4	4
	Klasės (boundary)		11	11
	Klasės (control)		16	16
	Operacijos			51
Viso:		34	119	199

4.1.2. Viešbučio kambario rezervacijos IS projektas

Remiantis ICONIX metodu sukurtas *MagicDraw* projektas tema „Viešbučio kambario rezervacijos informacinė sistema“. Kaip ir knygos autorių aprašytas „Internetinės knygų parduotuvės (angl. *Internet Bookstore*)“ pavyzdys, taip ir šis buvo kurtas etapais pradedant nuo reikalavimų, tęsiant preliminaraus projektavimo etapu ir t.t. Šiam „Viešbučio kambario rezervacijos informacinė sistema“ projektui buvo iškelti tokie pagrindiniai funkciniai reikalavimai:

1. Vartotojas, atlikęs viešbučio kambario rezervaciją, ją vis dar gali peržiūrėti ir redaguoti.
2. Vartotojas gali atšaukti viešbučio rezervaciją.
3. Vartotojas viešbučio kambarį gali rezervuoti internetu.
4. Naujas vartotojas gali užsiregistruoti informacinėje sistemoje, o užsiregistravęs vartotojas – gali prisijungti.
5. Vartotojas gali apmokėti viešbučio kambario rezervaciją.

Remiantis tokiais pagrindiniais reikalavimais buvo sumodeliuota informacinė sistema remiantis ICONIX metodu. Sumodeliavus sistemą, buvo atliekama *MagicDraw* projekto validacija darbe sukurtomis taisyklėmis, patikrinta, ar projekte neliko esminių klaidų norint užtikrinti modelio elementų suderinamumą. Projekte panaudoti elementų tipai, diagramos, pateikiamos lentelėje nr. 5 pagal atitinkamus etapus su juose panaudotais objektais.

5 lentelė. Viešbučio kambario rezervacijos IS projektą sudarantys elementai

Objektai \ Kriterijus		I etapo objektai	II etapo objektai	III etapo objektai
Diagramos	Klasių	1	1	1
	Panaudojimo atvejų	1	1	1
	Išbaigtumo		8	8
	Sekų			8
Elementai	Asociacijos	8	84	84
	Klasės (entity)	4	4	4
	Panaudojimo atvejai	8	8	8
	Aktoriai	1	1	2
	Klasės (boundary)		10	10
	Klasės (control)		9	9
	Operacijos			56
Viso:		23	126	191

4.1.3. Studentų kurti IS projektai

Kauno technologijos universitete dėstomame modulyje „Informacinių sistemų inžinerijos metodai ir modeliai“ studentai supažindinami su dažniausiai naudojamais informacinių sistemų kūrimo metodais ir modeliais, tokiais kaip *EKD* metodika, objektų-procesų metodika, unifikuotas procesas (angl. *Unified Process*), *MERODE* metodika ir kt. Taip pat buvo įtrauktas ir ICONIX procesas informacinėms sistemoms kurti. Teorinių paskaitų metu studentams buvo pristatytas ICONIX metodas, jo pagrindiniai bruožai, laboratorinio darbo užduotis buvo suprojektuoti sistemą naudojant ICONIX metodą – sudaryti organizacijos veiklos modelį, reikalavimų specifikaciją, projektą organizacijos informacinei sistemai taikant ICONIX metodą. Studentai buvo suskirstyti darbo grupėmis po 2-3 studentus grupėje ir grupinio darbo metu turėjo suprojektuoti sistemą naudojant ICONIX metodą. Visiems studentams buvo pateikta viena tema – „Renginių planavimas ir organizavimas“. Studentai galėjo pasirinkti, kurį sistemos modulį projektuos. Buvo galima pasirinkti projektuoti iš tokių mažesnių temos potemių: pakvietimai, renginio vieta, nakvynė, transportavimas, matinimas, pranešimų lektoriai ir kt. Laboratorinio darbo tikslas – renginių organizavimo proceso arba jo dalies kompiuterizavimas.

Norint patikrinti ICONIX metodui skirtų taisyklių veiksmingumą validacijai tikrinti ir elementų suderinamumui užtikrinti, buvo surinkti trys prieš tai aprašyto modulio studentų grupiniai darbai, kurti naudojant ICONIX metodą. Šių trijų projektų kiekybiniai kriterijai pateikti lentelėse nr. 6, 7 ir 8.

6 lentelė. Studentų projektą nr. 1 sudarantys elementai

Objektai \ Kriterijus		I etapo objektai	II etapo objektai	III etapo objektai
Diagramos	Klasių	1	2	2
	Panaudojimo atvejų	1	1	1
	Išbaigtumo		8	8
	Sekų			8
Elementai	Asociacijos	24	41	41
	Klasės (entity)	2	2	2
	Panaudojimo atvejai	9	9	9
	Aktoriai	1	1	1
	Klasės (boundary)		8	8
	Klasės (control)		2	2
	Operacijos			93
Viso:		38	74	175

7 lentelė. Studentų projektą nr. 2 sudarantys elementai

Objektai \ Kriterijus		I etapo objektai	II etapo objektai	III etapo objektai
Diagramos	Klasių	1	1	1
	Panaudojimo atvejų	1	1	1
	Išbaigtumo		4	4
	Sekų			4
Elementai	Asociacijos	8	50	50
	Klasės (entity)	1	2	2
	Panaudojimo atvejai	4	4	4
	Aktoriai	1	1	1
	Klasės (boundary)		4	4
	Klasės (control)		17	17
	Operacijos			31
Viso:		16	84	119

8 lentelė. Studentų projektą nr. 3 sudarantys elementai

Objektai \ Kriterijus		I etapo objektai	II etapo objektai	III etapo objektai
Diagramos	Klasių	2	2	2
	Panaudojimo atvejų	1	1	1
	Išbaigtumo		3	3
	Sekų			3
Elementai	Asociacijos	14	38	38
	Klasės (entity)	7	7	7
	Panaudojimo atvejai	3	3	3
	Aktoriai	1	1	1
	Klasės (boundary)		6	6
	Klasės (control)		8	8
	Operacijos			20
Viso:		28	69	92

Lentelėse nr. 6, 7 ir 8 pateikiami kiekybiniai studentų kurtų projektų kriterijai. Daugiausia elementų sukurta pirmajame projekte. Jų iš viso yra 175. Tarp jų yra 19 diagramų. Antrąjį studentų projektą sudaro 119 elementų, tarp kurių yra 10 diagramų. Trečiasis projektas kiekybiškai yra mažiausias. Jį sudaro tik 92 elementai, iš kurių 9 yra diagramos. Kaip matome, daugiausia elementų sudaro pirmąjį projektą ir jame yra beveik dvigubai daugiau elementų negu trečiajamą studentų projekte.

4.2. Eksperimento rezultatai

4.2.1. Knygų parduotuvės IS eksperimento rezultatai

Lentelėse 9, 10, 11 pateikiama detali informacija, kiek kokių elementų, diagramų sudaro knygoje pateiktą projektą. Į *MagicDraw* programos aplinką buvo perkeltos tik tos diagramos, kurios yra pilnai sukurtos ir aprašytos knygos autorių. Papildomų diagramų nebuvo sukurta, naudojami tik tie elementai, kurie aprašyti autorių. Panaudojimo atvejai, kuriems nebuvo sukurtos išbaigtumo arba sekų diagramos, nebuvo perkelti į programos aplinką. Suskaičiuoti elementai pateikiami priklausomai nuo etapo, taip pat pat pagal elementų rūšį.

Lentelėje 9 pateikiama ICONIX projekto I etapo diagramų ir elementų sudėtis. Pirmajame etape sukurtos dvi diagramos – klasių arba dalykinės srities, ir panaudojimo atvejų. Klasių diagramą sudaro 8 klasių elementai. Panaudojimo atvejų diagramą sudaro 5 panaudojimo atvejai, taip pat

keturi aktoriai. Abejose diagramose iš viso panaudota 15 asociacijų. Iš viso pirmajame etape sukurta 34 objektai, iš kurių yra 2 diagramos. Pirmajam etapui nėra sukurta validacijos taisyklių, sukurti elementai nebuvo tikrinti.

9 lentelė. Knygų parduotuvės IS projekto I etapas

Kriterijus / Objektai		Kiek **senų** objektų	Kiek **naujų** objektų	Kiek iš **viso** objektų etape	Kiek suveikė taisyklių	Suveikusių taisyklių numeriai
Diagramos	Klasių	-	1	1	-	-
	Panaudojimo atvejų	-	1	1		
Elementai	Asociacijos	-	15	15		
	Klasės	-	8	8		
	Panaudojimo atvejai	-	5	5		
	Aktoriai	-	4	4		
Viso:		-	34	34		

Kitoje 10-oje lentelėje pateikiama ICONIX projekto II etapo diagramų ir elementų sudėtis. Antrajame etape papildomai sukurtos jau 5 išbaigtumo diagramos. Antrajame etape iš viso jau yra sukurtos 7 diagramos. Klasių ir panaudojimo atvejų diagramų elementų sudėtis nesikeitė, o išbaigtumo diagramose sukurta iš viso 27 nauji elementai, o 8 klasių elementai perpanaudoti iš anksčiau sukurtos klasių diagramos. Taip pat buvo panaudotos 53 naujos asociacijos išbaigtumo diagramose. Iš viso antrame etape jau panaudotos 68 asociacijos, 35 klasės. Naujų elementų antrajame etape – 85. Iš viso elementų antrajame etape – 119.

Antrajam etapui buvo atlikta sukurtų taisyklių validacija, tačiau nei viena taisyklė nesuveikė. Knygoje pateiktas antro etapo pavyzdys yra korektiškas ir be klaidų.

10 lentelė. Knygų parduotuvės IS projekto II etapas

Kriterijus / Objektai		Kiek **senų** objektų	Kiek **naujų** objektų	Kiek iš **viso** objektų etape	Kiek suveikė taisyklių	Suveikusių taisyklių numeriai
Diagramos	Klasių	1	-	1	-	-
	Panaudojimo atvejų	1	-	1		
	Išbaigtumo	-	5	5		
Elementai	Asociacijos	15	53	68		
	Klasės (entity)	8	-	8		
	Panaudojimo atvejai	5	-	5		
	Aktoriai	4	-	4		
	Klasės (boundary)	-	11	11		
	Klasės (control)	-	16	16		
Viso:		34	85	119		

Toliau 11-oje lentelėje pateikiamas ICONIX projekto III etapo elementų sudėtis. Lentelėje vis dar matoma, kiek yra elementų iš praėjusio etapo, taip pat pateikta, kiek naujų elementų sukurta šiame etape. Nuo antro etapo diagramų skiltis papildyta sekų diagramų skiltimi. Trečiajame etape buvo sukurtos 5 sekų diagramos. Dabar projekte iš viso yra 12 diagramų. Taip pat buvo sukurta 24 naujų asociacijų – dabar asociacijų iš viso yra 92. Taip pat sekų diagramoje sukurta operacijų, kurių anksčiau nebuvo naudota. Naujų operacijų – 51. Iš viso buvo sukurta 80 naujų objektų, o iš viso trečiajame etape objektų yra 199, iš kurių 12 diagramų.

Šiame etape taip pat nesuveikė nei viena validacijos taisyklė, skirta trečiajam etapui tikrinti. D.Rosenberg‘o ir K.Scott‘o knygoje „Applying use case driven object modeling with UML“ pateiktas pavyzdys pilnai išpildytas ir korektiškas.

11 lentelė. Knygų parduotuvės IS projekto III etapas

Kriterijus / Objektai		Kiek **senų** objektų	Kiek **naujų** objektų	Kiek iš **viso** objektų etape	Kiek suveikė taisyklių	Suveikusių taisyklių numeriai
Diagramos	Klasių	1	-	1	-	-
	Panaudojimo atvejų	1	-	1		
	Išbaigtumo	5	-	5		
	Sekų	-	5	5		
Elementai	Asociacijos	68	24	92		
	Klasės (entity)	8	-	8		
	Panaudojimo atvejai	5	-	5		
	Aktoriai	4	-	4		
	Klasės (boundary)	11	-	11		
	Klasės (control)	16	-	16		
	Operacijos	-	51	51		
Viso:		119	80	199		

4.2.2. Viešbučio kambario rezervacijos IS eksperimento rezultatai

Lentelėje nr. 12 pateikta pirmojo etapo diagramų ir elementų sudėtis. Pirmajame etape sukurtos dvi diagramos – klasių arba dalykinės srities, ir panaudojimo atvejų. Klasių diagramą sudaro 4 klasių elementai. Panaudojimo atvejų diagramą sudaro 8 panaudojimo atvejai, taip pat vienas aktorius. Abejose diagramose iš viso panaudotos 8 asociacijos. Iš viso pirmajame etape sukurti 23 objektai, iš kurių yra 2 diagramos. Pirmajam etapui nėra sukurta validacijos taisyklių, sukurti elementai nebuvo tikrinti.

12 lentelė. „Viešbučio“ IS I etapas

Kriterijus / Objektai		Kiek senų objektų	Kiek naujų objektų	Kiek iš viso objektų etape	Kiek suveikė taisyklių	Suveikusių taisyklių numeriai
Diagramos	Klasių	-	1	1	-	-
	Panaudojimo atvejų	-	1	1		
Elementai	Asociacijos	-	8	8		
	Klasės	-	4	4		
	Panaudojimo atvejai	-	8	8		
	Aktoriai	-	1	1		
Viso:		-	23	23		

Kitoje 13-oje lentelėje pateikiama projekto II etapo diagramų ir elementų sudėtis. Antrajame etape papildomai sukurtos jau 8 išbaigtumo diagramos. Antrajame etape iš viso jau yra sukurta 10 diagramų. Klasių ir panaudojimo atvejų diagramų elementų sudėtis nesikeitė, o išbaigtumo diagramose sukurta iš viso 19 naujų klasių elementų, o 8 klasių elementai perpanaudoti iš anksčiau sukurtos klasių diagramos. Taip pat buvo panaudotos 76 naujos asociacijos išbaigtumo diagramose. Iš viso antrame etape jau panaudotos 84 asociacijos, 23 klasės. Naujų elementų antrajame etape – 103. Iš viso elementų antrajame etape – 126.

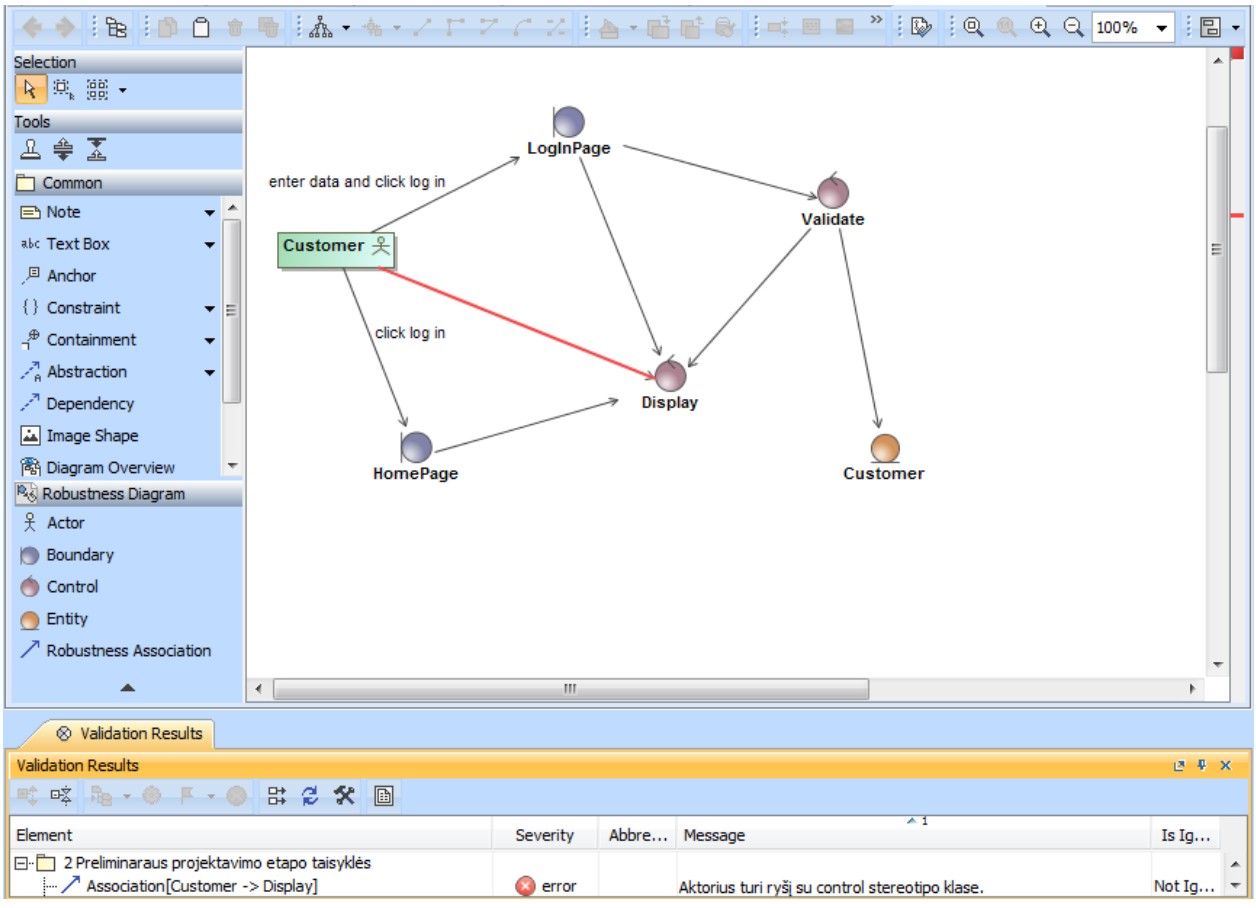
Antrajam etapui buvo atlikta sukurtų taisyklių validacija, suveikė dvi skirtingos taisyklės. Abi taisyklės suveikė po vieną kartą. Suveikė taisyklės, pažymėtos numeriais 2.2 ir 2.6.

Abi taisyklės tikrina ryšius išbaigtumo diagramose, kurie jungia šių diagramų elementus. Detalesnę informaciją, kokios taisyklės suveikė, galite peržiūrėti 7-ame priedų skyriuje lentelėje nr. 26.

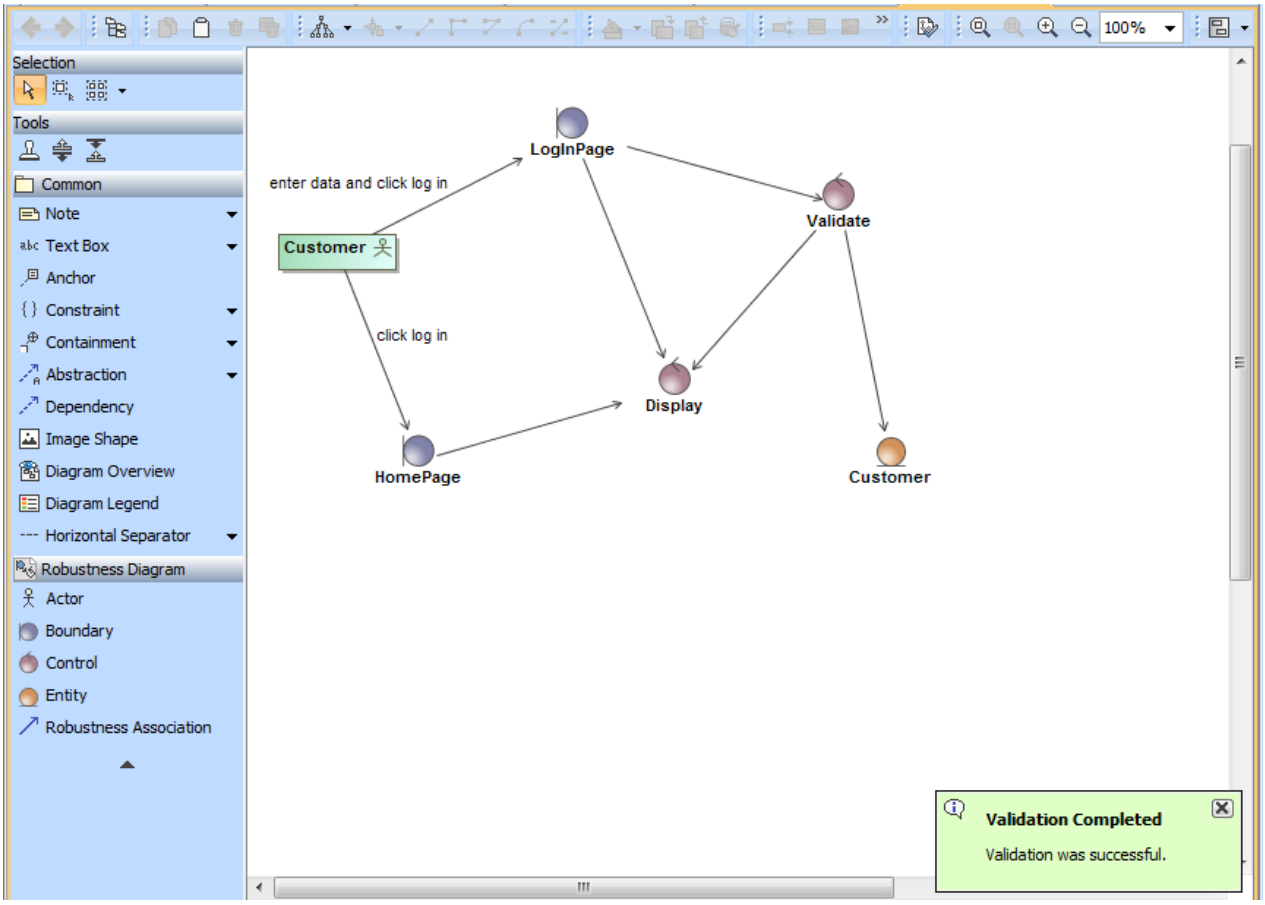
13 lentelė. „Viešbučio“ IS II etapas

Objektai \ Kriterijus		Kiek **senų** objektų	Kiek **naujų** objektų	Kiek iš **viso** objektų etape	Kiek skirtingų taisyklių suveikė	Kiek iš viso kartų suveikė taisyklės	Suveikusių taisyklių numeriai
Diagramos	Klasių	1	-	1	2	2	2.2. 2.6.
	Panaudojimo atvejų	1	-	1			
	Išbaigtumo	-	8	8			
Elementai	Asociacijos	8	76	84			
	Klasės (entity)	4	-	4			
	Panaudojimo atvejai	8	-	8			
	Aktoriai	1	-	1			
	Klasės (boundary)	-	10	10			
	Klasės (control)	-	9	9			
Viso:		23	103	126			

Paveiksle 4.1 pateiktas *MagicDraw* programos ekrano vaizdas, kai suveikė taisyklės atliekant antrojo etapo diagramų validaciją. Pavyzdyje suveikusi taisyklė tikrina, ar tarp aktoriaus ir valdiklio stereotipo klasės yra ryšys. Pagal ICONIX metodą, tarp minėtų elementų ryšio negali būti, todėl taisyklė suveikė, kad ryšys tarp objektų nubrėžtas neteisingai. Paveiksle 4.2 pateiktas programos ekrano vaizdas po klaidos ištaisymo, atlikus validaciją programa pateikia lentelę apie sėkmingai atliktą patikrinimą.



4.1 pav. „Viešbučio IS“ II etapo suveikusios taisyklės



4.2 pav. „Viešbučio IS“ II etapo validacija po pataisymo

Toliau 14-oje lentelėje pateikiama projekto III etapo elementų sudėtis. Lentelėje vis dar matoma, kiek yra elementų iš praėjusio etapo, taip pat pateikta, kiek naujų elementų sukurta šiame etape. Nuo antro etapo diagramų skiltis papildyta sekų diagramų skiltimi. Trečiajame etape buvo sukurtos 8 sekų diagramos. Dabar projekte iš viso yra 18 diagramų. Taip pat sekų diagramose sukurta operacijų, kurių anksčiau nebuvo naudota. Naujų operacijų – 56. Iš viso buvo sukurta 64 nauji objektai, o iš viso trečiajame etape objektų yra 191, iš kurių 18 diagramų.

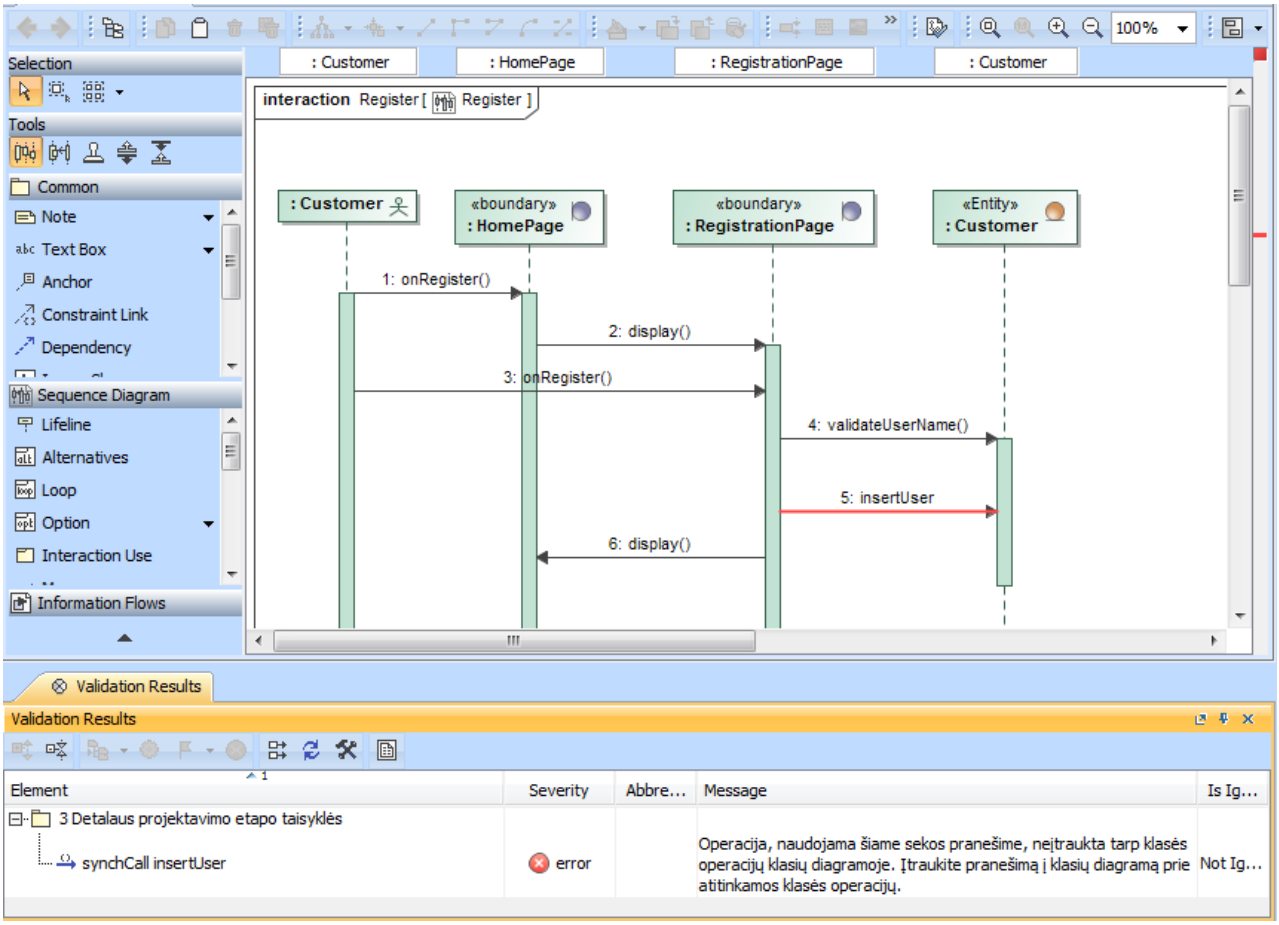
Trečiajam etapui buvo atlikta sukurtų taisyklių validacija, suveikė trys skirtingos taisyklės. Taisyklės iš viso suveikė 11 kartų. Suveikė taisyklės, pažymėtos numeriais 3.1, 3.4 ir 3.5. Šios taisyklės tikrina sekų diagramų operacijas, ar šios įtrauktos į klasių diagramą prie atitinkamos klasės operacijų, taip pat tikrinama, ar išbaigtumo diagramoje panaudotos ribinės klasės sekų diagramoje panaudotos kaip gyvavimo linijos, ar visiem panaudojimo atvejam sukurta po sekų diagramą. Detalesnę informaciją, kokios tiksliai taisyklės suveikė, kiek kartų kiekviena suveikė galite peržiūrėti 7-ame priedo skyriuje lentelėje nr. 26.

14 lentelė. „Viešbučio“ IS III etapas

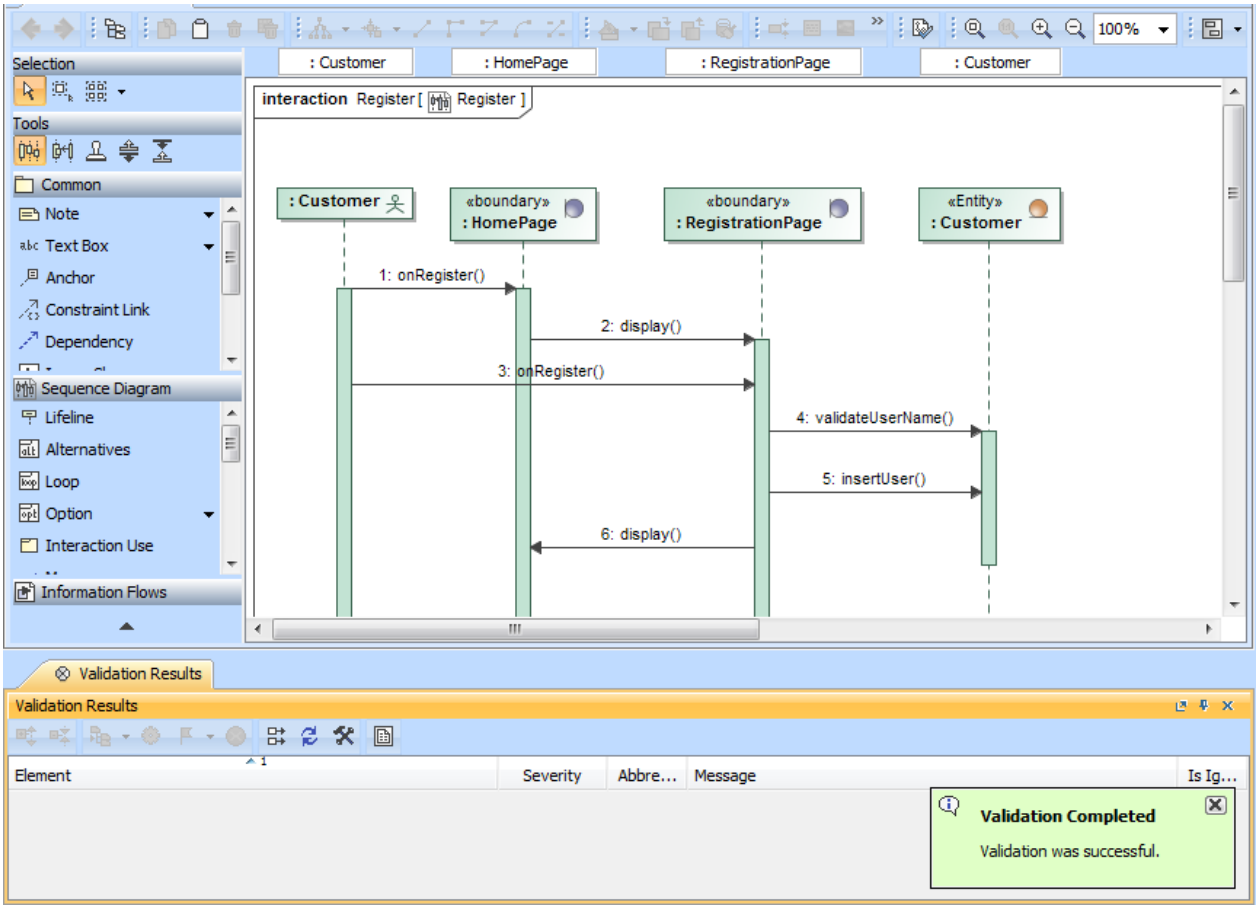
Kriterijus / Objektai		Kiek **senų** objektų	Kiek **naujų** objektų	Kiek iš **viso** objektų etape	Kiek skirtingų taisyklių suveikė	Kiek iš viso kartų suveikė taisyklės	Suveikusių taisyklių numeriai
Diagramos	Klasių	1	-	1	3	11	3.1. 3.4. 3.5.
	Panaudojimo atvejų	1	-	1			
	Išbaigtumo	8	-	8			
	Sekų	-	8	8			
Elementai	Asociacijos	84	-	84			
	Klasės (entity)	4	-	4			

Objektai \ Kriterijus		Kiek **senų** objektų	Kiek **naujų** objektų	Kiek iš **viso** objektų etape	Kiek skirtingų taisyklių suveikė	Kiek iš viso kartų suveikė taisyklės	Suveikusių taisyklių numeriai
	Panaudojimo atvejai	8	-	8			
	Aktoriai	2	-	2			
	Klasės (boundary)	10	-	10			
	Klasės (control)	9	-	9			
	Operacijos	-	56	56			
Viso:		127	64	191			

Paveiksle 4.3 pateiktas programos *MagicDraw* ekrano vaizdas, kai suveikė trečio etapo validacijos taisyklės. Paveiksle matoma „Redaguoti kambario rezervaciją“ sekų diagrama, kurioje suveikė taisyklė, tikrinanti sekų diagramų pranešimus, ar šie įtraukti klasių diagramoje. Kitame 4.4 paveiksle pateikta ta pati diagrama po pataisymų. Operacijos buvo įtrauktos į klasių diagramą ir atlikus validaciją gautas pranešimas, kad klaidų validuojant trečiąjį etapą neaptikta.



4.3 pav. „Viešbučio IS“ III etapo suveikusios taisyklės



4.4 pav. „Viešbučio IS“ III etapo validacija po pataisymo

4.2.3. Studentų projekto nr. 1 eksperimento rezultatai

Lentelėje nr. 15 pateikta pirmojo etapo diagramų ir elementų sudėtis. Pirmajame etape sukurtos dvi diagramos – klasių arba dalykinės srities, ir panaudojimo atvejų. Klasių diagramą sudaro 2 klasių elementai. Panaudojimo atvejų diagramą sudaro 9 panaudojimo atvejai, taip pat vienas aktorius. Abejose diagramose iš viso panaudotos 24 asociacijos. Iš viso pirmajame etape sukurti 38 objektai, iš kurių yra 2 diagramos. Pirmajam etapui nėra sukurta validacijos taisyklių, sukurti elementai nebuvo tikrinti.

15 lentelė. Pirmo studentų projekto I etapas

Kriterijus / Objektai		Kiek **senų** objektų	Kiek **naujų** objektų	Kiek iš **viso** objektų etape	Kiek suveikė taisyklių	Suveikusių taisyklių numeriai
Diagramos	Klasių	-	1	1	-	-
	Panaudojimo atvejų	-	1	1		
Elementai	Asociacijos	-	24	24		
	Klasės	-	2	2		
	Panaudojimo atvejai	-	9	9		
	Aktoriai	-	1	1		
Viso:		-	38	38		

Kitoje 16-oje lentelėje pateikiama projekto II etapo diagramų ir elementų sudėtis. Antrajame etape papildomai sukurtos jau 8 išbaigtumo diagramos, taip pat nauja klasių diagrama.

Antrajame etape iš viso jau yra sukurta 11 diagramų. Klasių ir panaudojimo atvejų diagramų elementų sudėtis nesikeitė, o išbaigtumo diagramose sukurta iš viso 10 naujų klasių elementų, o 2 klasių elementai perpanaudoti iš anksčiau sukurtos klasių diagramos. Taip pat buvo panaudota 17 naujų asociacijų išbaigtumo diagramose. Iš viso antrame etape jau panaudota 41 asociacija, 12 klasių. Naujų elementų antrajame etape – 36. Iš viso elementų antrajame etape – 74.

Antrajam etapui buvo atlikta sukurtų taisyklių validacija, suveikė dvi skirtingos taisyklės. Abi taisyklės suveikė po ne vieną kartą. Suveikė taisyklės, pažymėtos numeriais 2.1 ir 2.4. Viena iš taisyklių tikrina, ar kiekvienam panaudojimo atvejui yra sukurta po išbaigtumo diagramą. Kita taisyklė tikrina, ar nėra ryšių tarp esybių stereotipo klasių. Detalesnę informaciją, kokios taisyklės ir kiek kartų suveikė, galite peržiūrėti 7-ame priedo skyriuje lentelėje nr. 27.

16 lentelė. Pirmo studentų projekto II etapas

Kriterijus / Objektai		Kiek **senų** objektų	Kiek **naujų** objektų	Kiek iš **viso** objektų etape	Kiek skirtingų taisyklių suveikė	Kiek iš viso kartų suveikė taisyklės	Suveikusių taisyklių numeriai
Diagramos	Klasių	1	1	2	2	10	2.1. 2.4.
	Panaudojimo atvejų	1	-	1			
	Išbaigtumo	-	8	8			
Elementai	Asociacijos	24	17	41			
	Klasės (entity)	2	-	2			
	Panaudojimo atvejai	9	-	9			
	Aktoriai	1	-	1			
	Klasės (boundary)	-	8	8			
	Klasės (control)	-	2	2			
Viso:		38	36	74			

Toliau 17-oje lentelėje pateikiama projekto III etapo elementų sudėtis. Lentelėje vis dar matoma, kiek yra elementų iš praėjusio etapo, taip pat pateikta, kiek naujų elementų sukurta šiame etape. Nuo antro etapo diagramų skiltis papildyta sekų diagramų skiltimi. Trečiajame etape buvo sukurtos 8 sekų diagramos. Dabar projekte iš viso yra 19 diagramų. Taip pat sekų diagramoje sukurta operacijų, kurių anksčiau nebuvo naudota. Naujų operacijų – 93. Iš viso buvo sukurtas 101 naujas objektas, o iš viso trečiajame etape objektų yra 175, iš kurių 19 diagramų.

Trečiajam etapui buvo atlikta sukurtų taisyklių validacija, suveikė dvi skirtingos taisyklės. Taisyklės iš viso suveikė 29 kartus. Suveikė taisyklės, pažymėtos numeriais 3.1 ir 3.4. Šios taisyklės tikrina sekų diagramų operacijas, ar šios įtrauktos į klasių diagramą prie atitinkamos klasės operacijų, taip pat tikrinama, ar visiem panaudojimo atvejam sukurta po sekų diagramą. Detalesnę informaciją, kokios tiksliai taisyklės suveikė, kiek kartų kiekviena suveikė galite peržiūrėti 7-ame priedo skyriuje lentelėje nr. 27.

17 lentelė. Pirmo studentų projekto III etapas

Kriterijus / Objektai		Kiek **senų** objektų	Kiek **naujų** objektų	Kiek iš **viso** objektų etape	Kiek skirtingų taisyklių suveikė	Kiek iš viso kartų suveikė taisyklės	Suveikusių taisyklių numeriai
Diagramos	Klasių	2	-	2	2	29	3.1. 3.4.
	Panaudojimo atvejų	1	-	1			
	Išbaigtumo	8	-	8			
	Sekų	-	8	8			
Elementai	Asociacijos	41	-	41			

Objektai \ Kriterijus		Kiek **senų** objektų	Kiek **naujų** objektų	Kiek iš **viso** objektų etape	Kiek skirtingų taisyklių suveikė	Kiek iš viso kartų suveikė taisyklės	Suveikusių taisyklių numeriai
	Klasės (entity)	2	-	2			
	Panaudojimo atvejai	9	-	9			
	Aktoriai	1	-	1			
	Klasės (boundary)	8	-	8			
	Klasės (control)	2	-	2			
	Operacijos	-	93	93			
Viso:		74	101	175			

4.2.4. Studentų projekto nr. 2 eksperimento rezultatai

Lentelėje nr. 18 pateikta pirmojo etapo diagramų ir elementų sudėtis. Pirmajame etape sukurtos dvi diagramos – klasių arba dalykinės srities, ir panaudojimo atvejų. Klasių diagramą sudaro 1 klasės elementas. Panaudojimo atvejų diagramą sudaro 4 panaudojimo atvejai, taip pat vienas aktorius. Abejose diagramose iš viso panaudotos 8 asociacijos. Iš viso pirmajame etape sukurta 16 objektų, iš kurių yra 2 diagramos. Pirmajam etapui nėra sukurta validacijos taisyklių, sukurti elementai nebuvo tikrinti.

18 lentelė. Antro studentų projekto I etapas

Objektai \ Kriterijus		Kiek **senų** objektų	Kiek **naujų** objektų	Kiek iš **viso** objektų etape	Kiek suveikė taisyklių	Suveikusių taisyklių numeriai
Diagramos	Klasių	-	1	1	-	-
	Panaudojimo atvejų	-	1	1		
Elementai	Asociacijos	-	8	8		
	Klasės	-	1	1		
	Panaudojimo atvejai	-	4	4		
	Aktoriai	-	1	1		
Viso:		-	16	16		

Kitoje 19-oje lentelėje pateikiama projekto II etapo diagramų ir elementų sudėtis. Antrajame etape papildomai sukurtos jau 4 išbaigtumo diagramos. Antrajame etape iš viso jau yra sukurtos 6 diagramos. Klasių ir panaudojimo atvejų diagramų elementų sudėtis nesikeitė, o išbaigtumo diagramose sukurtas iš viso 21 naujas klasių elementas, taip pat sukurtas vienas naujas klasės objektas su esybės stereotipu, o kita esybės stereotipo klasė panaudota iš pirmo etapo sukurtų elementų. Taip pat buvo panaudotos 42 naujos asociacijos išbaigtumo diagramose. Iš viso antrame etape jau panaudota 50 asociacijų, 23 klasės. Naujų elementų antrajame etape – 68. Iš viso elementų antrajame etape – 84.

Antrajam etapui buvo atlikta sukurtų taisyklių validacija, suveikė dvi skirtingos taisyklės. Abi taisyklės suveikė po ne vieną kartą. Suveikė taisyklės, pažymėtos numeriais 2.1 ir 2.3. Viena iš taisyklių tikrina, ar kiekvienam panaudojimo atvejui yra sukurta po išbaigtumo diagramą. Kita taisyklė tikrina, ar nėra ryšių tarp esybių ir/ar ribinių stereotipo klasių. Detalesnę informaciją, kokios taisyklės ir kiek kartų suveikė, galite peržiūrėti 7-ame priedo skyriuje lentelėje nr. 28.

19 lentelė. Antro studentų projekto II etapas

Kriterijus / Objektai		Kiek **senų** objektų	Kiek **naujų** objektų	Kiek iš **viso** objektų etape	Kiek skirtingų taisyklių suveikė	Kiek iš viso kartų suveikė taisyklės	Suveikusių taisyklių numeriai
Diagramos	Klasių	1	-	1	2	6	2.1. 2.3.
	Panaudojimo atvejų	1	-	1			
	Išbaigtumo	-	4	4			
Elementai	Asociacijos	8	42	50			
	Klasės (entity)	1	1	2			
	Panaudojimo atvejai	4	-	4			
	Aktoriai	1	-	1			
	Klasės (boundary)	-	4	4			
	Klasės (control)	-	17	17			
Viso:		16	68	84			

Toliau 20-oje lentelėje pateikiama projekto III etapo elementų sudėtis. Lentelėje vis dar matoma, kiek yra elementų iš praėjusio etapo, taip pat pateikta, kiek naujų elementų sukurta šiame etape. Nuo antro etapo diagramų skiltis papildyta sekų diagramų skiltimi. Trečiajame etape buvo sukurtos 4 sekų diagramos. Dabar projekte iš viso yra 10 diagramų. Taip pat sekų diagramoje sukurta operacijų, kurių anksčiau nebuvo naudota. Naujų operacijų – 31. Iš viso buvo sukurti 35 nauji objektai, o iš viso trečiajame etape objektų yra 119, iš kurių 10 diagramų.

Trečiajam etapui buvo atlikta sukurtų taisyklių validacija, suveikė trys skirtingos taisyklės. Taisyklės iš viso suveikė 11 kartų. Suveikė taisyklės, pažymėtos numeriais 3.1, 3.5 ir 3.6. Šios taisyklės tikrina sekų diagramų operacijas, ar šios įtrauktos į klasių diagramą prie atitinkamos klasės operacijų, taip pat tikrinama, ar išbaigtumo diagramoje panaudotos ribinės ir esybės klasės sekų diagramoje kaip gyvavimo linijos. Detalesnę informaciją, kokios tiksliai taisyklės suveikė, kiek kartų kiekviena suveikė, galite peržiūrėti 7-ame priedo skyriuje lentelėje nr. 28.

20 lentelė. Antro studentų projekto III etapas

Kriterijus / Objektai		Kiek **senų** objektų	Kiek **naujų** objektų	Kiek iš **viso** objektų etape	Kiek skirtingų taisyklių suveikė	Kiek iš viso kartų suveikė taisyklės	Suveikusių taisyklių numeriai
Diagramos	Klasių	1	-	1	3	34	3.1. 3.5. 3.6.
	Panaudojimo atvejų	1	-	1			
	Išbaigtumo	4	-	4			
	Sekų	-	4	4			
Elementai	Asociacijos	50	-	50			
	Klasės (entity)	2	-	2			
	Panaudojimo atvejai	4	-	4			
	Aktoriai	1	-	1			
	Klasės (boundary)	4	-	4			
	Klasės (control)	17	-	17			
	Operacijos	-	31	31			
Viso:		84	35	119			

4.2.5. Studentų projekto nr. 3 eksperimento rezultatai

Lentelėje nr. 21 pateikta pirmojo etapo diagramų ir elementų sudėtis. Pirmajame etape sukurtos trys diagramos – klasių ir dalykinės srities, taip pat panaudojimo atvejų. Klasių diagramas sudaro 7 klasių elementai. Panaudojimo atvejų diagramą sudaro 3 panaudojimo atvejai, taip pat vienas aktorius. Diagramose iš viso panaudota 14 asociacijų. Iš viso pirmajame etape sukurti 28 objektai, iš kurių yra 3 diagramos. Pirmajam etapui nėra sukurta validacijos taisyklių, sukurti elementai nebuvo tikrinti.

21 lentelė. Trečio studentų projekto I etapas

Kriterijus / Objektai		Kiek **senų** objektų	Kiek **naujų** objektų	Kiek iš **viso** objektų etape	Kiek suveikė taisyklių	Suveikusių taisyklių numeriai
Diagramos	Klasių	-	2	2	-	-
	Panaudojimo atvejų	-	1	1		
Elementai	Asociacijos	-	14	14		
	Klasės	-	7	7		
	Panaudojimo atvejai	-	3	3		
	Aktoriai	-	1	1		
Viso:		-	28	28		

Sekančioje 22-oje lentelėje pateikiama projekto II etapo diagramų ir elementų sudėtis. Antrajame etape papildomai sukurtos jau 3 išbaigtumo diagramos. Antrajame etape iš viso jau yra sukurtos 6 diagramos. Klasių ir panaudojimo atvejų diagramų elementų sudėtis nesikeitė, o išbaigtumo diagramose sukurta iš viso 14 naujų klasių elementų, o esybės stereotipo klasės panaudotos iš pirmo etapo sukurtų elementų. Taip pat buvo panaudotos 24 naujos asociacijos išbaigtumo diagramose. Iš viso antrame etape jau panaudota 38 asociacijų, 21 klasė. Naujų elementų antrajame etape – 41. Iš viso elementų antrajame etape – 69.

Antrajam etapui buvo atlikta sukurtų taisyklių validacija, suveikė dvi skirtingos taisyklės. Abi taisyklės suveikė po ne vieną kartą. Suveikė taisyklės, pažymėtos numeriais 2.1 ir 2.6. Viena iš taisyklių tikrina, ar kiekvienam panaudojimo atvejui yra sukurta po išbaigtumo diagramą. Kita taisyklė tikrina, ar nėra ryšių tarp aktorių ir valdiklio stereotipo klasių. Detalesnę informaciją, kokios taisyklės ir kiek kartų suveikė, galite peržiūrėti 7-ame priedo skyriuje lentelėje nr. 29.

22 lentelė. Trečio studentų projekto II etapas

Kriterijus / Objektai		Kiek **senų** objektų	Kiek **naujų** objektų	Kiek iš **viso** objektų etape	Kiek skirtingų taisyklių suveikė	Kiek iš viso kartų suveikė taisyklės	Suveikusių taisyklių numeriai
Diagramos	Klasių	2	-	2	2	4	2.1. 2.6.
	Panaudojimo atvejų	1	-	1			
	Išbaigtumo	-	3	3			
Elementai	Asociacijos	14	24	38			
	Klasės (entity)	7	-	7			
	Panaudojimo atvejai	3	-	3			
	Aktoriai	1	-	1			
	Klasės (boundary)	-	6	6			
	Klasės (control)	-	8	8			
Viso:		28	41	69			

Toliau 23-oje lentelėje pateikiama projekto III etapo elementų sudėtis. Lentelėje vis dar matoma, kiek yra elementų iš praėjusio etapo, taip pat pateikta, kiek naujų elementų sukurta šiame etape. Nuo antro etapo diagramų skiltis papildyta sekų diagramų skiltimi. Trečiajame etape buvo sukurtos 3 sekų diagramos. Dabar projekte iš viso yra 9 diagramos. Taip pat sekų diagramoje sukurta operacijų, kurių anksčiau nebuvo naudota. Naujų operacijų – 20. Iš viso buvo sukurti 23 nauji objektai, o iš viso trečiajame etape objektų yra 92, iš kurių 9 diagramos.

Trečiajam etapui buvo atlikta sukurtų taisyklių validacija, suveikė keturios skirtingos taisyklės. Taisyklės iš viso suveikė 28 kartus. Suveikė taisyklės, pažymėtos numeriais 3.1, 3.2, 3.4 ir 3.5. Šios taisyklės tikrina sekų diagramų operacijas, ar šios įtrauktos į klasių diagramą prie atitinkamos klasės operacijų, taip pat tikrinama, ar išbaigtumo diagramoje panaudotos ribinės ir esybės klasės sekų diagramoje panaudotos kaip gyvavimo linijos, ar sekų diagramoje panaudota gyvavimo linija yra įtraukta klasių diagramoje. Detalesnę informaciją, kokios tiksliai taisyklės suveikė, kiek kartų kiekviena suveikė galite peržiūrėti 7-ame priedo skyriuje lentelėje nr. 29.

23 lentelė. Trečio studentų projekto III etapas

Kriterijus / Objektai		Kiek **senų** objektų	Kiek **naujų** objektų	Kiek iš **viso** objektų etape	Kiek skirtingų taisyklių suveikė	Kiek iš viso kartų suveikė taisyklės	Suveikusių taisyklių numeriai
Diagramos	Klasių	2	-	2	4	28	3.1. 3.2. 3.5. 3.6.
	Panaudojimo atvejų	1	-	1			
	Išbaigtumo	3	-	3			
	Sekų	-	3	3			
Elementai	Asociacijos	38	-	38			
	Klasės (entity)	7	-	7			
	Panaudojimo atvejai	3	-	3			
	Aktoriai	1	-	1			
	Klasės (boundary)	6	-	6			
	Klasės (control)	8	-	8			
	Operacijos	-	20	20			
Viso:		69	23	92			

4.3. Sprendimo veikimo ir savybių analizė, kokybės kriterijų įvertinimas

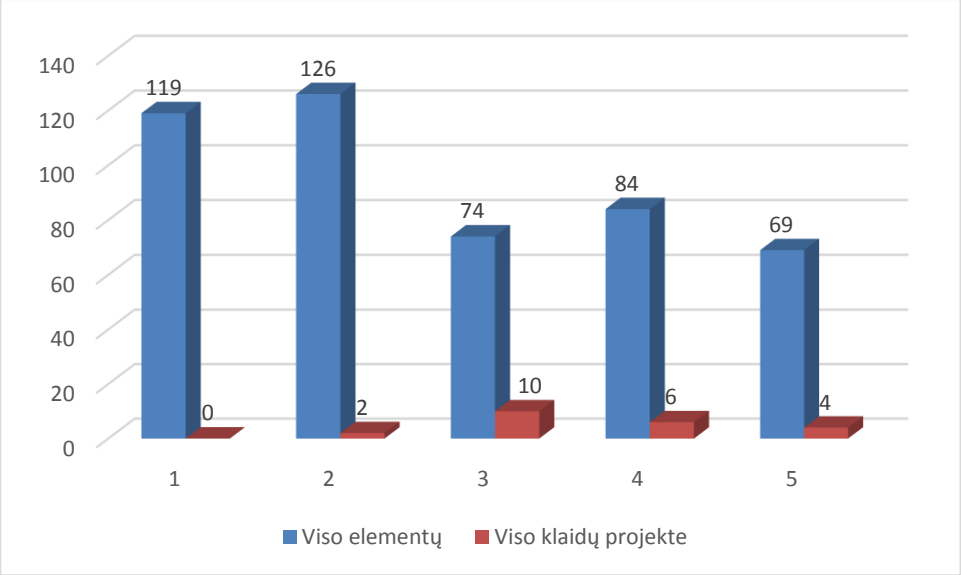
Atlikus visų projektų suderinamumo patikrinimą pritaikius sukurtų taisyklių rinkinį, galima teigti, kad taisyklės suveikia - aptinka klaidas, kai jų yra. Sukurtos taisyklės tikrina skirtingus metamodelio elementus – klases, asociacijas, gyvavimo linijas ir kt., kurie yra dažniausiai naudojami kuriant projektus taikant ICONIX metodą. Pritaikant taisykles, galima užtikrinti geresnį projekto elementų suderinamumą, kad elementai naudojami tinkamai pagal numatytas metodo taisykles. Pažeidus kurią nors taisyklę – *MagicDraw* programa informuoja, kurioje vietoje aptikta klaida, kaip ją būtų galima ištaisyti.

Iš atlikto eksperimento galima teigti, kad dažniausiai projektų antrajame etape suveikianti taisyklė yra 2.1, kuri tikrina, ar panaudojimo atvejui yra sukurta išbaigtumo diagrama. Projektuose, kuriuose aptikta ši klaida, išbaigtumo diagrama buvo kuriama naudojant klasių diagramą, nors yra specialiai pritaikyta išbaigtumo tipo (angl. *robustness*) diagrama. Detalesnė informacija, kaip dažnai ir kokia taisyklė suveikdavo antrajame projektų etape, pateikiama lentelėje nr. 24.

24 lentelė. Antro etapo aptiktų klaidų kiekis projektuose

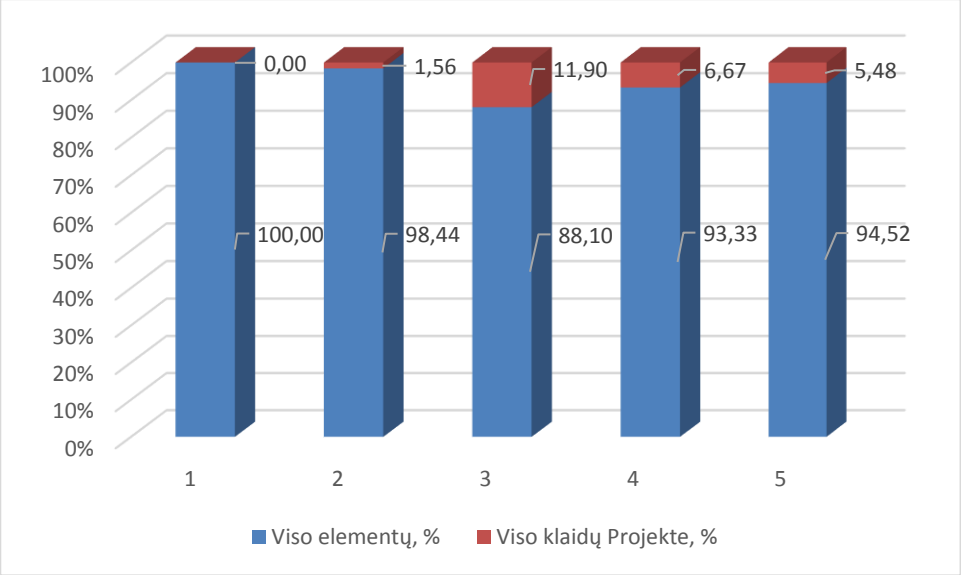
Taisyklė / Projektas	Viso elementų	2.1.	2.2.	2.3.	2.4.	2.5.	2.6.	Viso klaidų projekte
I	119	-	-	-	-	-	-	-
II	126	-	1	-	-	-	1	2
III	74	9	-	-	1	-	-	10
IV	84	4	-	2	-	-	-	6
V	69	3	-	-	-	-	1	4
Viso		16	1	2	1	-	2	

Taip pat paveiksle 4.5 grafiškai pateikta, kiek elementų sukurta ir tikrinama, ir kiek rasta klaidų antrajame projekto etape. Diagramoje pateikta visų penkių projektų informacija, kiek elementų sudaro antrąjį tikrinamą etapą ir kiek klaidų aptikta atliekant elementų suderinamumo patikrinimą. Pirmojo projekto antrąjį etapą sudaro 119 elementų, tarp jų neaptikta nei viena klaida. Antrąjį projektą sudaro 126 elementai, tarp jų aptiktos 2 klaidos. Trečiojo projekto antrąjį etapą sudaro 74 elementai, tarp jų aptikta 10 klaidų. Ketvirtojo projekto antrąjį etapą sudaro 84 elementai, tarp jų – 6 klaidos. Paskutinįjį penktąjį projektą sudaro 69 elementai, iš jų 4 aptikti klaidingi.



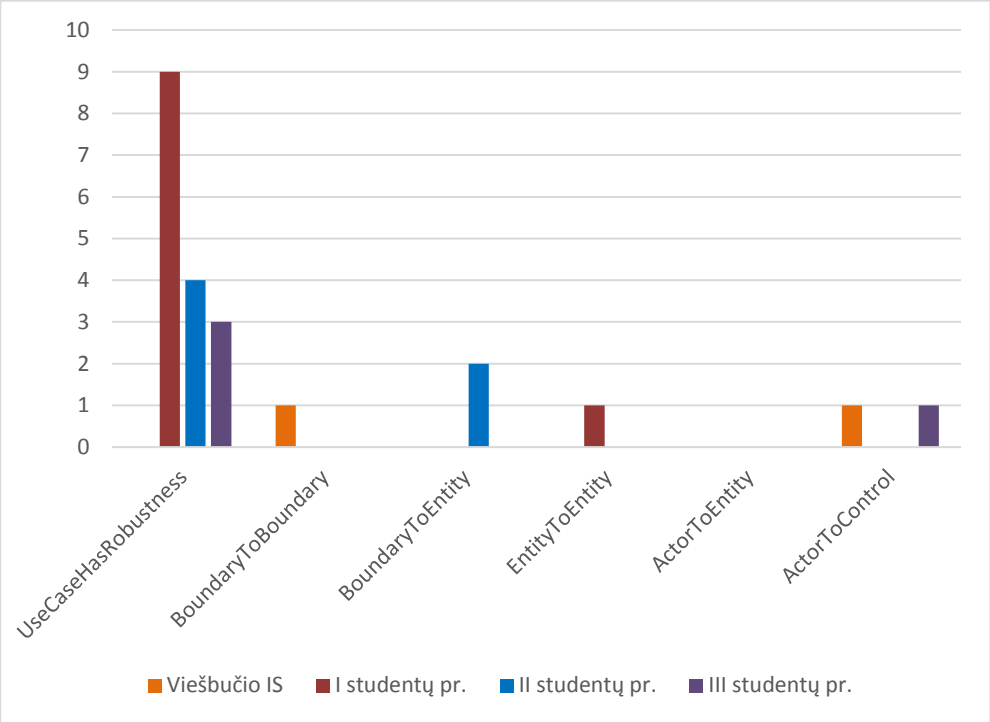
4.5 pav. Antro etapo elementų ir klaidų pasiskirstymas

Taip pat projektų antrus etapus sudarančių elementų ir aptiktų klaidų santykis pateiktas paveiksle 4.6. Šiame paveiksle galima matyti, kad daugiausiai klaidų santykinai aptikta trečiajame projekte, kuriame tik 88.10 proc. projekto yra be klaidų. Šis projektas yra vienas iš studentų darbų. Mažiausiai klaidų aptikta projekte „Viešbučio kambarių rezervacijos IS“, kuriame net 98.44 proc. projekto elementų yra teisingai panaudota. Pirmajame projekte „Internetinė knygų IS“ neaptikta nei viena elementų suderinimo klaida.



4.6 pav. Antro etapo elementų ir klaidų santykis, %

Paveiksle nr. 4.7 pateikiama diagrama, kurioje vaizduojamas antrajame etape aptiktų klaidų kiekis. Diagramoje pateikiamos visos 6 antrojo etapo taisyklių rinkinio taisyklės, ir prie kiekvienos iš jų vaizduojama, kuriame projekte kurios taisyklės pažeidimas buvo aptiktas. Diagramoje vaizduojamos klaidos iš keturių projektų, kuriuose buvo aptikta bent viena klaida atliekant antrojo etapo taisyklių rinkinio taikymą. Projektas, kuriame nebuvo aptikta klaidų, šioje diagramoje neįtrauktas.



4.7 pav. Antro etapo aptiktų klaidų kiekis projektuose

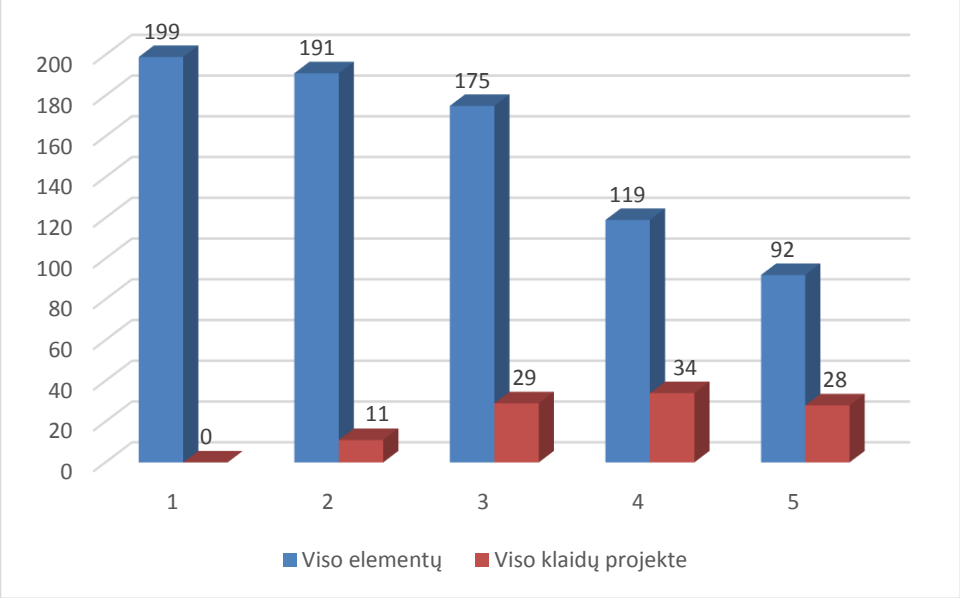
Atliktame eksperimente taip pat buvo tikrinamas ir trečio etapo elementų suderinamumas taip, kaip ir antrojo etapo. Trečiajame projektų etape dažniausiai suveikianti taisyklė yra 3.1, kuri tikrina, ar sekų diagramoje tarp gyvavimo linijų naudojamas pranešimas yra įtrauktas prie atitinkamos klasės klasių diagramoje. Taisyklės suveikimo dažnumą ir pasikartojimą projektuose galima peržiūrėti lentelėje nr. 27. Šioje lentelėje pateikiami taisyklių numeriai, žymintys atitinkamas

taisykles, taip pat pirmame stulpelyje sužymėti projektai nuo pirmo iki penkto, galima peržiūrėti taisyklės suveikimą tiek projekte, tiek jos suveikimo dažnumą. Klaida, kai suveikia 3.1 taisyklė, visuose projektuose pasitaikė dažniausiai. Taip pat trečiame projekto kūrimo etape dažnai suveikdavo taisyklės 3.5 ar 3.6, kurios tikrina, ar visos išbaigtumo diagramose panaudotos ribinės ir esybės klasės yra panaudotos sekų diagramose kaip gyvavimo linijos. Detalesnė informacija, kaip dažnai ir kokia taisyklė suveikdavo trečiajame projektų etape, pateikiama lentelėje nr. 25.

25 lentelė. Trečio etapo aptiktų klaidų kiekis projektuose

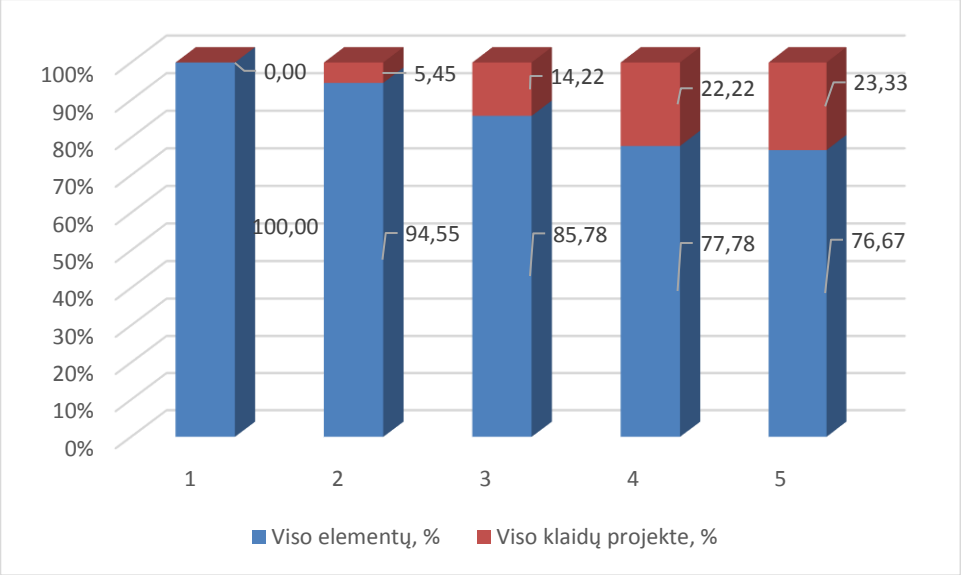
Taisyklė / Projektas	Viso elementų	3.1.	3.2.	3.3.	3.4.	3.5.	3.6.	Viso klaidų projekte:
I	199	-	-	-	-	-	-	-
II	191	9	-	-	1	1	-	11
III	175	28	-	-	1	-	-	29
IV	119	28	-	-	-	3	3	34
V	92	20	1	-	-	6	1	28
Viso klaidų:		85	1	-	2	10	4	

Taip pat paveiksle 4.8 grafiškai pateikta, kiek elementų sukurta ir tikrinama, ir kiek rasta klaidų trečiajame projekto etape. Diagramoje pateikta visų penkių projektų informacija, kiek elementų sudaro trečiąjį tikrinamą etapą ir kiek klaidų aptikta atliekant elementų suderinamumo patikrinimą. Pirmojo projekto trečiąjį etapą sudaro 199 elementai, tarp jų neaptikta nei viena klaida. Antrąjį projektą sudaro 191 elementas, tarp jų aptikta 11 klaidų. Trečiojo projekto trečiąjį etapą sudaro 175 elementai, tarp jų aptiktos 29 klaidos. Ketvirtojo projekto trečiąjį etapą sudaro 119 elementų, tarp jų – 34 klaidos. Paskutinįjį penktąjį projektą sudaro 92 elementai, iš jų 28 aptikti klaidingi.



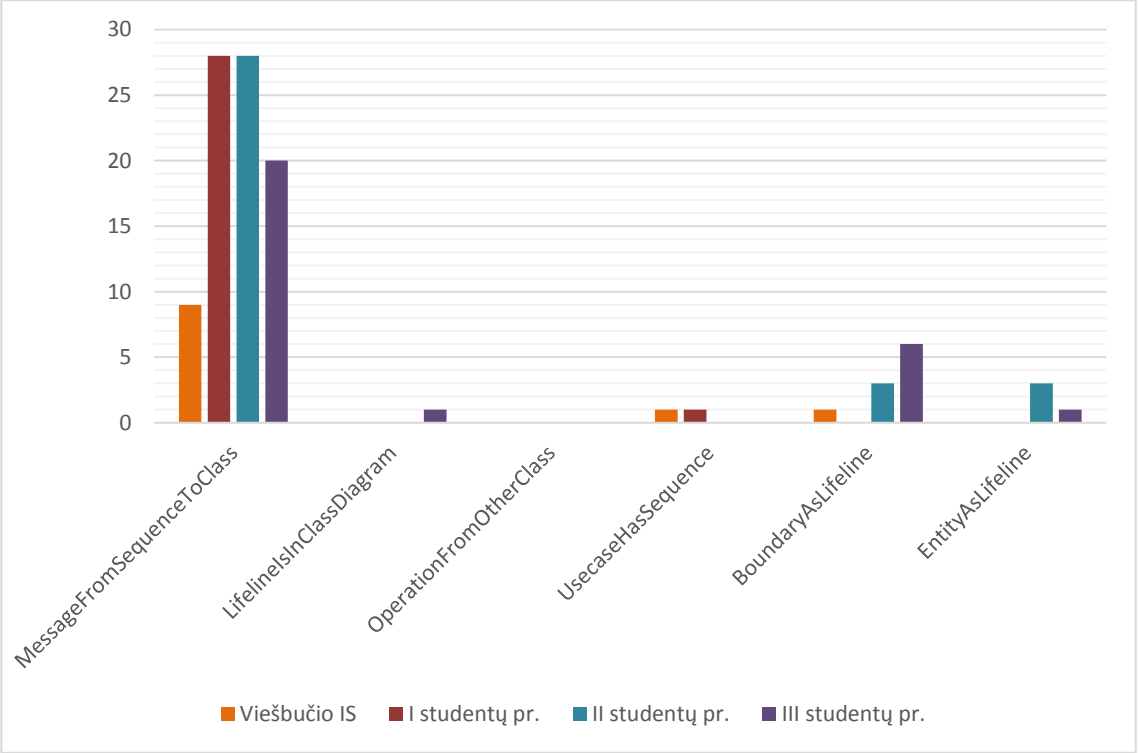
4.8 pav. Trečio etapo elementų ir klaidų pasiskirstymas

Taip pat projektų trečius etapus sudarančių elementų ir aptiktų klaidų santykis pateiktas paveiksle 4.9. Šiame paveiksle galima matyti, kad daugiausiai klaidų santykinai aptikta penktajame projekte, kuriame tik 76.67 proc. projekto yra be klaidų. Šis projektas yra vienas iš studentų darbų. Mažiausiai klaidų aptikta projekte „Viešbučio kambarių rezervacijos IS“, kuriame net 94.55 proc. projekto elementų yra teisingai panaudoti. Pirmajame projekte „Internetinė knygų IS“ neaptikta nei viena elementų suderinimo klaida.



4.9 pav. Trečio etapo elementų ir klaidų santykis, %

Paveiksle nr. 4.10 pateikiama diagrama, kurioje vaizduojamas trečiajame etape aptiktų klaidų kiekis. Diagramoje pateikiamos visos 6 trečiojo etapo taisyklių rinkinio taisyklės, ir prie kiekvienos iš jų vaizduojama, kuriame projekte kurios taisyklės pažeidimas buvo aptiktas. Diagramoje vaizduojamos klaidos iš keturių projektų, kuriuose buvo aptikta bent viena klaida atliekant trečiojo etapo taisyklių rinkinio taikymą. Projektas, kuriame nebuvo aptikta klaidų, šioje diagramoje neįtrauktas.



4.10 pav. Trečio etapo aptiktų klaidų kiekis projektuose

4.4. Sprendimo taikymo rekomendacijos

Atliekant sprendimo taikymą pastebėta, kad sukurtos modelio suderinimo taisyklės gali būti taikomos tiek dideliem projektam, tiek mažesniem. Atliekant elementų suderinimo patikrinimą svarbiausia pasirinkti tinkamą projekto modelio vietą, nes taisyklės skirtos tik nurodytam modelio

elementų etapui tikrinti. Jeigu bus tikrinamas visas projektas, arba kitas etapas, negu numatyta taisyklėms tikrinti, gali būti gautos klaidos, kur jų iš tikrųjų nėra. Taisyklės yra pritaikytos tik tam tikram etapui, pavyzdžiui, detalaus projektavimo, todėl šio etapo taisykles reikėtų taikyti tik tame etape sukurtoms diagramoms, elementams tikrinti. Jeigu bus pasirinktas kitas etapas ir pritaikytos minėto etapo suderinimo tikrinimo taisyklės, bus gautos klaidos, nes keičiantis etapams, keičiasi ir taisyklių galiojimo vieta. Norint nesuklysti, kada kokios taisyklės, jų paketai turėtų būti taikomi, reikėtų vadovautis darbe pateiktomis rekomendacijomis, kurios aprašytos 3.2 skyriuje, taip pat grafiškai pateiktos 3.24 paveiksle. Rekomendacijose ir 3.24 paveiksle nurodyta ir aprašyta, kurios taisyklės kuriam etapui pritaikytos, kaip rekomenduojama kurti projektą programoje, kad vėliau būtų galima lengviau pritaikyti suderinimo taisykles. Pavyzdžiui, rekomendacijose nurodyta, kad trečiojo – detalaus projektavimo – etapo sukurtos diagramos turėtų būti patalpintos viename aplanke, kurį pasirinkus pritaikomos trečiojo etapo suderinimo taisyklės. Laikantis rekomendacijose nurodytų reikalavimų, lengviau galima atlikti suderinimo patikrinimą, reikia atlikti mažiau veiksmų, kad būtų pasiektas geriausias rezultatas. Aprašyme pateiktos tik rekomendacijos, kaip kurti projektą, kad lengviau būtų galima pritaikyti suderinimo taisykles, tačiau galima projektą kurti ir skirstyti kaip vartotojui patogiau, tik vėliau norint atlikti elementų suderinimo patikrinimą, reikės atlikti daugiau veiksmų tam, kad būtų pasirinktos reikiamos diagramos, kurioms bus taikomos taisyklės. Kaip jau anksčiau minėta, taisyklės gali suveikti ir nekorektiškai, jeigu pasirinktas neteisingas variantas – tiek taisyklių, tiek diagramų, elementų. Taigi apibendrinant, geriausia būtų laikytis aprašytų rekomendacijų, t. y. kuriant projektą laikytis aplankų struktūros, kurti tikslias reikiamas diagramas, tik tada galima užtikrinti geriausią taisyklių veikimą. Sukūrus netikslią struktūrą, netikslias diagramas, taisykles bus sudėtinga pritaikyti, jos suveiks netiksliai, gali būti, kad suveiks ne tose vietose, kur turėtų aptikti klaidas. Taip pat šios taisyklės pritaikytos tik anksčiau minėtiems atvejams – ICONIX metodu kuriamiems modeliams norint užtikrinti metodo pagrindinius principus skirtinguose modelio kūrimo etapuose, taip pat norint užtikrinti pagrindinių UML naudojamų elementų ir diagramų tarpusavio ryšius. Taisyklių rinkinį rekomenduojama taikyti, kai norima tiksliai modeliuoti kuriamos programinės įrangos projektą, tada galima užtikrinti taisyklingesnį diagramų kūrimą ir griežtesnį ICONIX metodo taisyklių laikymąsi.

5. REZULTATŲ APIBENDRINIMAS IR IŠVADOS

1. Atlikus analizę nustatyta, kad tie patys UML modelio elementai yra naudojami skirtingose diagramose ir įrankiai neužtikrina visiško elementų ir diagramų suderinamumo, todėl norint sugriežtinti elementų suderinamumą reikalingos taisyklės.
2. Išanalizavus mokslinius straipsnius UML modelio suderinimo srityje nustatyta, kad darbuose akcentuojami formalūs arba apribojimais grindžiami metodai, kurie nėra pritaikyti jokiam programinės įrangos kūrimo metodui, o šiame darbe sukurtas taisyklių rinkinys pritaikytas ICONIX metodu kuriamų modelių diagramų elementų suderinamumui tikrinti.
3. Darbe pasiūlytas taisyklių rinkinys, apimantis dviejų diagramų tipų taisykles: sekų ir klasių, skirtas UML elementų ir jų ryšių tikrinimui, taip pat ir specifiniam ICONIX metodui pritaikytas taisykles, apimančias išbaigtumo diagramas, jų elementų ryšius.
4. Pasiūlytas taisyklių rinkinys, apimantis ir ICONIX metodu kurtų modelių validaciją, realizuotas kaip *MagicDraw* įrankio validacijos modulio dalis, kuri papildo standartines *MagicDraw* programos validacijos taisyklių dalis.
5. Pasiūlytas taisyklių rinkinys eksperimentiškai ištirtas taisykles pritaikant keliem skirtingiem projektų variantam ir nustatyta, kad tai palengvina dažniausiai pasitaikančių klaidų radimą, o tai padeda tobulinti kuriamą modelį.
6. Eksperimentiškai ištyrus ir projektams pritaikius validacijos taisykles nustatyta, kad tiek studentų darbuose, tiek kituose projektuose iš ICONIX specifinių taisyklių dažniausiai klaidos buvo aptinkamos taisyklėje, tikrinančioje klasių panaudojimą sekų ir išbaigtumo diagramose, o iš UML elementų suderinimo dažniausiai klaidos buvo aptinkamos sekų diagramose, kai pranešimai nepanaudojami kaip operacijos klasių diagramoje.
7. Ateityje šis taisyklių rinkinys gali būti patobulintas papildant jį naujomis taisyklėmis, tikrinančiomis tiek ICONIX metodo, tiek UML elementus ar diagramas, taip pat kaip ir sukuriant automatišką pagrindinių taisyklių pritaikymą *MagicDraw* programoje.

6. LITERATŪRA

[1] G. Booch, J. Rumbaugh ir I. Jacobson, The Unified Modeling Language User Guide, 2nd mont., Massachusetts: Addison Wesley Professional, 2005.

[2] I. Object Management Group, „Documents Associated with Unified Modeling Language," 12 2015. [Tinkle]. Available: http://www.omg.org/spec/UML/2.5/PDF/.

[3] K. Hamilton ir R. Miles, Learning UML, 2.0 mont., B. M. a. M. T. O'Brien, Mont., Sebastopol: O'Reilly Media, Inc., 2006.

[4] D. Rosenberg ir M. Stephens, Use Case Driven Object Modeling with UML Theory and Practice, J. Gennick, Mont., New York: Apress, 2007.

[5] I. Object Management Group, „Object Constraint Language," 12 2015. [Tinkle]. Available: http://www.omg.org/spec/OCL/2.4/PDF/.

[6] C. Jordi, „Object Constraint Language (OCL) tutorial," Internet Interdisciplinary Institute , 21 03 2012. [Tinkle]. Available: http://modeling-languages.com/ocl-tutorial/. [Kreiptasi 11 10 2016].

[7] D. B. Demuth, „OCL (Object Constraint Language) by Example," Technische Universitat Dresden, 2009. [Tinkle]. Available: https://st.inf.tu-dresden.de/files/general/OCLByExampleLecture.pdf. [Kreiptasi 15 10 2016].

[8] „MagicDraw UserManual," No Magic, Inc, 2015. [Tinkle]. Available: http://www.nomagic.com/files/manuals/MagicDraw UserManual.pdf. [Kreiptasi 01 2016].

[9] „UML PROFILING AND DSL," No Magic, Inc, 2015. [Tinkle]. Available: https://www.nomagic.com/files/manuals/MagicDraw%20UMLProfiling&DSL%20UserGuide.pdf. [Kreiptasi 01 01 2016].

[10] W. Liu, S. M. Easterbrook ir M. J, „Rule-based detection of inconsistency in uml models," įtraukta *5th International Conference on Software Engineering*, London, 2002.

[11] H. Rasch ir H. Wehrheim, „Checking Consistency in UML Diagrams: Classes and State Machines. Formal Methods for Open Object-Based Distributed Systems," įtraukta *LNCS 2884*, 2003.

[12] F. Mokhati, P. Gagnon and M. Badri, "Verifying UML Diagrams With Model Checking: A Rewriting LogicBased Approach," in *Seventh International Conference on Quality Software (QSIC)*, Portland, 2007.

[13] K. E. Miloudi, Y. E. Amrani ir E. A., „An automated Translation of UML Class Diagrams into a Formal Specification to Detect UML Inconsistencies," įtraukta *Sixth International Conference on Software Engineering Advances (ICSEA 2001)*, Barcelona, 2011.

[14] F. L. Kotulski, „Assurance of system consistency during independent creation of UML diagrams," įtraukta *Proc. of the International Conference on Dependability of Computer Systems*, Szklarska Poreba, 2007.

[15] Z. Wang, „Ontology Based Semantics Cheking fo UML Activity Model," *Information Technology Journal*, t. 11, pp. 301-306, no 3. 2012.

[16] I. Object Management Group, „Unified Modeling Language," 01 2016. [Tinkle]. Available: http://www.omg.org/docs/formal/10-05-05.pdf.

[17] D. Chiorean, M. Pasca, A. Carcu, C. Botiza ir S. Moldovan, „Ensuring UML models consistency using the OCL Environment," p. 11.

[18] E. Pakalnickienė ir L. Nemuraitė, „Checking of conceptual models with integrity constraints," *Information Technology and Control*, t. 36, nr. no. 3, pp. 285-294, 2007.

[19] B. Hnatkowska, Z. Huzar ir J. Magott, „Consistency Checking in UML Models," p. 6.

[20] Z. Chen ir G. Motet, „A Language-Theoretic View on Guidelines and Consistency Rules of

UML,“ įtraukta *LNCS 5562*, 2009.

[21] P. G. Sapna ir H. Mohanty, „Ensuring consistency in relational repository of UML models,“ įtraukta *Proc. of the 10th International Conference of Information Technology*, Rouekela, 2007.

[22] J. Chanda, „Tranceability of Requirements and Consistency Verification of UML UseCase, Activity and Class diagram: A Formal Approatch,“ įtraukta *Proc. of International Conference on Methods and Models in Computer Science (ICM2C2 09)*, New Delhi, 2009.

[23] R. Dubauskaite ir O. Vasilecas, „Method on Specifying Consistency Rules among Different Aspect Models, expressed in UML,“ *Elektronika ir elektrotechnika*, p. 5, 2013.

[24] R. Hazra ir D. Shouvik, „Consistency between Use Case, Sequence nad Timing Diagram for Real Time Software Systems,“ *International Journal of Computer Applications*, p. 7, 2014.

[25] D. Rosenberg ir K. Scott, Applying Use Case Driven Object Modeling with UML, United States of America: Addison-Wesley, 2001.

7. PRIEDAI

7.1. Projektų validacijos rezultatai

26 lentelė. Viešbučio IS taisyklių suveikimo dažnumas

Taisyklė / Etapas	2.1. Kiekviena m panaudoji mo atvejui sudaroma detalizuota išbaigtumo diagrama.	2.2. Klasės su ribiniu (angl. boundary) stereotipu negali turėti ryšio su kitomis ribinėmis (angl. boundary) klasėmis išbaigtumo diagramoje.	2.3. Klasės su ribiniu (angl. boundary) stereotipu negali turėti ryšio su esybės (angl. entity) klasėmis išbaigtumo diagramoje.	2.4. Klasės su esybės (angl. entity) stereotipu negali turėti ryšio su esybės (angl. entity) klasėmis išbaigtumo diagramoje.	2.5. Aktoriai negali turėti ryšio su esybės (angl. entity) stereotipo klasėmis išbaigtumo diagramoje.	2.6. Aktoriai negali turėti ryšio su valdiklio (angl. control) stereotipo klasėmis išbaigtumo diagramoje.	3.1. Kiekviena operacija, įtraukta į sekų diagramos pranešimą, turi būti apibrėžta klasių diagramoje.	3.2. Kiekviena sekų diagramos gyvavimo linija (angl. lifeline) turi būti klasės egzemplior ius (angl. class) klasės diagramoje.	3.3. Kiekvienai klasei, kurioje operacijos naudoja kitą klasę, privalo būti priklausom ybės ryšys tarp šių klasių klasių diagramoje.	3.4. Kiekviena m panaudoji mo atvejui sudaroma detalizuota sekų diagrama.	3.5. Kiekvienas išbaigtumo diagramos ribinės (angl. boundary) klasės egzemplior ius turi būti panaudotas kaip sekų diagramos gyvavimo linija (angl. lifeline).	3.6. Kiekvienas išbaigtumo diagramos esybės (angl. entity) klasės egzemplior ius turi būti panaudotas kaip sekų diagramos gyvavimo linija (angl. lifeline).
1 etapas												
2 etapas	-	1	-	-	-	1						
3 etapas							9	-	-	1	1	-

27 lentelė. Pirmas studentų projektas taisyklių suveikimo dažnumas

Taisyklė / Etapas	2.1. Kiekviena m panaudoji mo atvejui sudaroma detalizuota išbaigtumo diagrama.	2.2. Klasės su ribiniu (angl. boundary) stereotipu negali turėti ryšio su kitomis ribinėmis (angl. boundary) klasėmis išbaigtumo diagramoje.	2.3. Klasės su ribiniu (angl. boundary) stereotipu negali turėti ryšio su esybės (angl. entity) klasėmis išbaigtumo diagramoje.	2.4. Klasės su esybės (angl. entity) stereotipu negali turėti ryšio su esybės (angl. entity) klasėmis išbaigtumo diagramoje.	2.5. Aktoriai negali turėti ryšio su esybės (angl. entity) stereotipo klasėmis išbaigtumo diagramoje.	2.6. Aktoriai negali turėti ryšio su valdiklio (angl. control) stereotipo klasėmis išbaigtumo diagramoje.	3.1. Kiekviena operacija, įtraukta į sekų diagramos pranešimą, turi būti apibrėžta klasių diagramoje.	3.2. Kiekviena sekų diagramos gyvavimo linija (angl. lifeline) turi būti klasės egzemplior ius (angl. class) klasės diagramoje.	3.3. Kiekvienai klasei, kurioje operacijos naudoja kitą klasę, privalo būti priklausom ybės ryšys tarp šių klasių klasių diagramoje.	3.4. Kiekviena m panaudoji mo atvejui sudaroma detalizuota sekų diagrama.	3.5. Kiekvienas išbaigtumo diagramos ribinės (angl. boundary) klasės egzemplior ius turi būti panaudotas kaip sekų diagramos gyvavimo linija (angl. lifeline).	3.6. Kiekvienas išbaigtumo diagramos esybės (angl. entity) klasės egzemplior ius turi būti panaudotas kaip sekų diagramos gyvavimo linija (angl. lifeline).
1 etapas												
2 etapas	9	-	-	1	-	-						
3 etapas							28	-	-	1	-	-

28 lentelė. Antras studentų projektas taisyklių suveikimo dažnumas

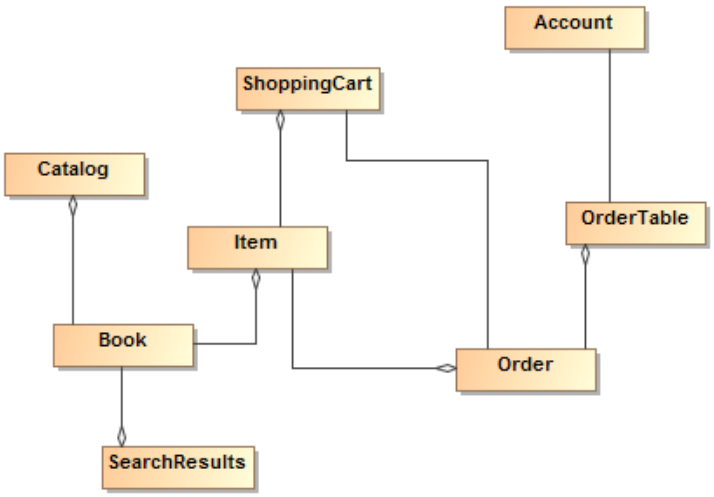
Taisyklė / Etapas	2.1. Kiekviena m panaudoji mo atvejui sudaroma detalizuota išbaigtumo diagrama.	2.2. Klasės su ribiniu (angl. boundary) stereotipu negali turėti ryšio su kitomis ribinėmis (angl. boundary) klasėmis išbaigtumo diagramoje.	2.3. Klasės su ribiniu (angl. boundary) stereotipu negali turėti ryšio su esybės (angl. entity) klasėmis išbaigtumo diagramoje.	2.4. Klasės su esybės (angl. entity) stereotipu negali turėti ryšio su esybės (angl. entity) klasėmis išbaigtumo diagramoje.	2.5. Aktoriai negali turėti ryšio su esybės (angl. entity) stereotipo klasėmis išbaigtumo diagramoje.	2.6. Aktoriai negali turėti ryšio su valdiklio (angl. control) stereotipo klasėmis išbaigtumo diagramoje.	3.1. Kiekviena operacija, įtraukta į sekų diagramos pranešimą, turi būti apibrėžta klasių diagramoje.	3.2. Kiekviena sekų diagramos gyvavimo linija (angl. lifeline) turi būti klasės egzemplior ius (angl. class) klasės diagramoje.	3.3. Kiekvienai klasei, kurioje operacijos naudoja kitą klasę, privalo būti priklausom ybės ryšys tarp šių klasių klasių diagramoje.	3.4. Kiekviena m panaudoji mo atvejui sudaroma detalizuota sekų diagrama.	3.5. Kiekvienas išbaigtumo diagramos ribinės (angl. boundary) klasės egzemplior ius turi būti panaudotas kaip sekų diagramos gyvavimo linija (angl. lifeline).	3.6. Kiekvienas išbaigtumo diagramos esybės (angl. entity) klasės egzemplior ius turi būti panaudotas kaip sekų diagramos gyvavimo linija (angl. lifeline).
1 etapas												
2 etapas	4	-	2	-	-	-						
3 etapas							28	-	-	-	3	3

29 lentelė. Trečias studentų projektas taisyklių suveikimo dažnumas

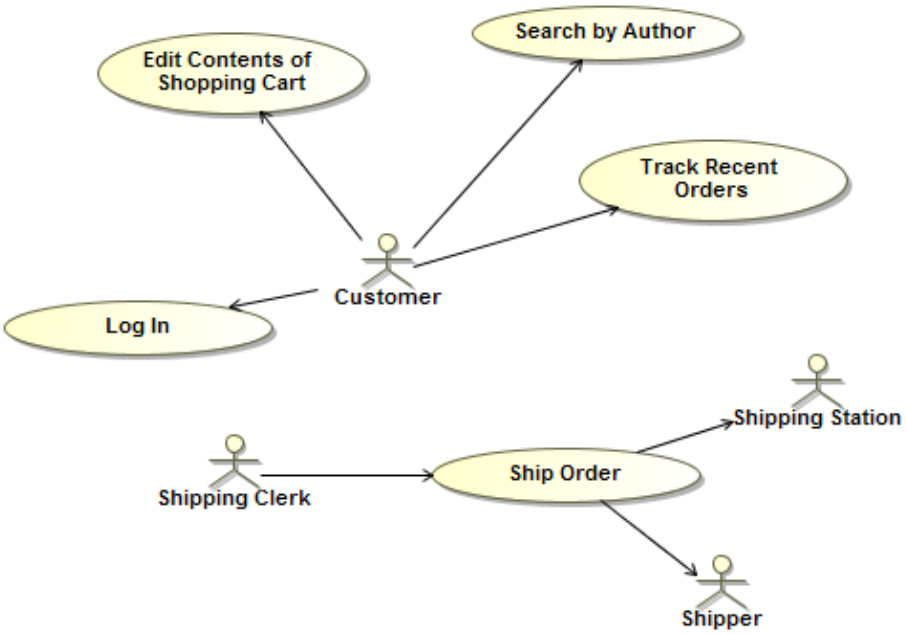
Taisyklė / Etapas	2.1. Kiekviena m panaudoji mo atvejui sudaroma detalizuota išbaigtumo diagrama.	2.2. Klasės su ribiniu (angl. boundary) stereotipu negali turėti ryšio su kitomis ribinėmis (angl. boundary) klasėmis išbaigtumo diagramoje.	2.3. Klasės su ribiniu (angl. boundary) stereotipu negali turėti ryšio su esybės (angl. entity) klasėmis išbaigtumo diagramoje.	2.4. Klasės su esybės (angl. entity) stereotipu negali turėti ryšio su esybės (angl. entity) klasėmis išbaigtumo diagramoje.	2.5. Aktoriai negali turėti ryšio su esybės (angl. entity) stereotipo klasėmis išbaigtumo diagramoje.	2.6. Aktoriai negali turėti ryšio su valdiklio (angl. control) stereotipo klasėmis išbaigtumo diagramoje.	3.1. Kiekviena operacija, įtraukta į sekų diagramos pranešimą, turi būti apibrėžta klasių diagramoje.	3.2. Kiekviena sekų diagramos gyvavimo linija (angl. lifeline) turi būti klasės egzemplior ius (angl. class) klasės diagramoje.	3.3. Kiekvienai klasei, kurioje operacijos naudoja kitą klasę, privalo būti priklausom ybės ryšys tarp šių klasių klasių diagramoje.	3.4. Kiekviena m panaudoji mo atvejui sudaroma detalizuota sekų diagrama.	3.5. Kiekvienas išbaigtumo diagramos ribinės (angl. boundary) klasės egzemplior ius turi būti panaudotas kaip sekų diagramos gyvavimo linija (angl. lifeline).	3.6. Kiekvienas išbaigtumo diagramos esybės (angl. entity) klasės egzemplior ius turi būti panaudotas kaip sekų diagramos gyvavimo linija (angl. lifeline).
1 etapas												
2 etapas	3	-	-	-	-	1						
3 etapas							20	1	-	-	6	1

7.2. Internetinės knygų parduotuvės IS duomenys

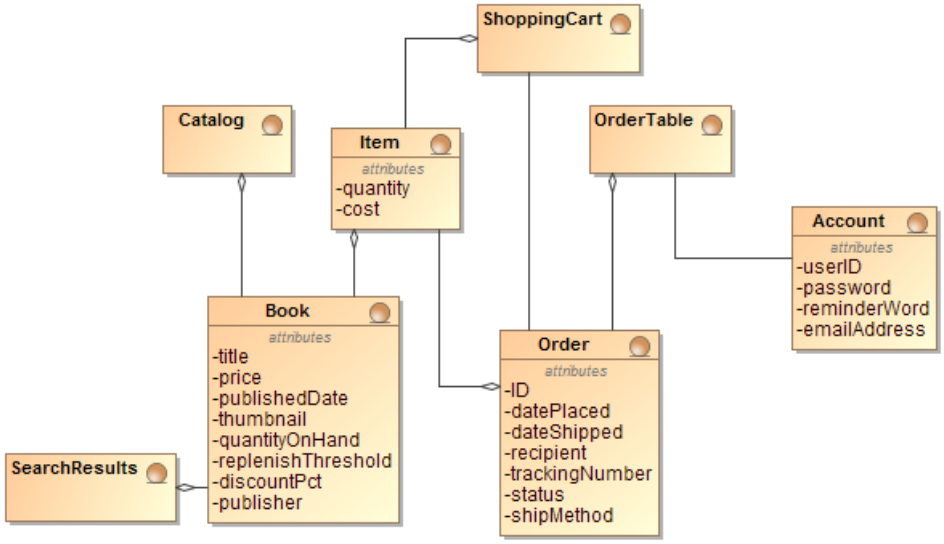
Reikalavimų etapas



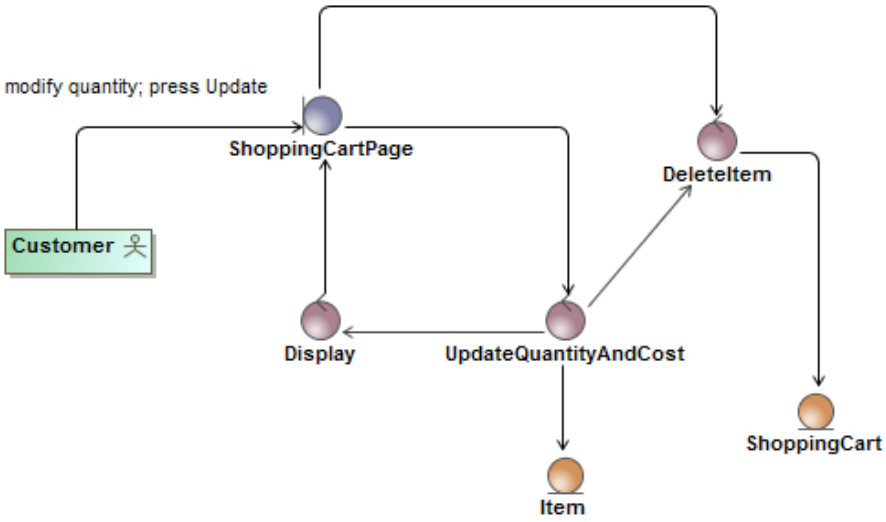
7.1 pav. Dalykinės srities modelis „Internetinė parduotuvė“



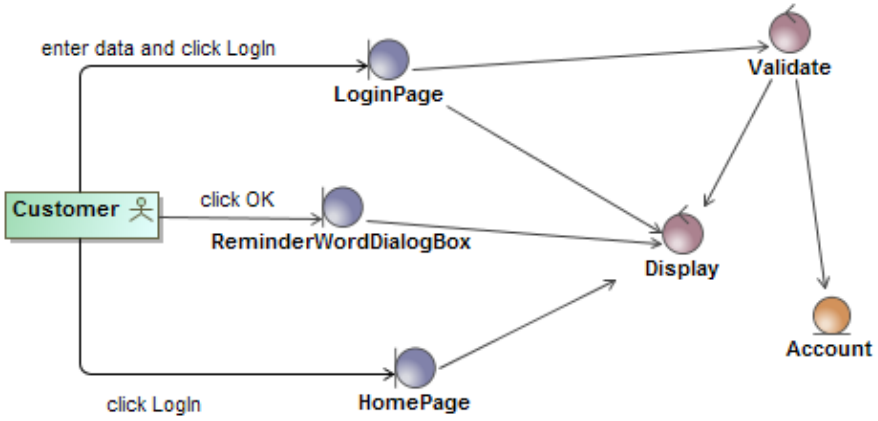
7.2 pav. Panaudojimo atvejų diagrama „Internetinė parduotuvė“



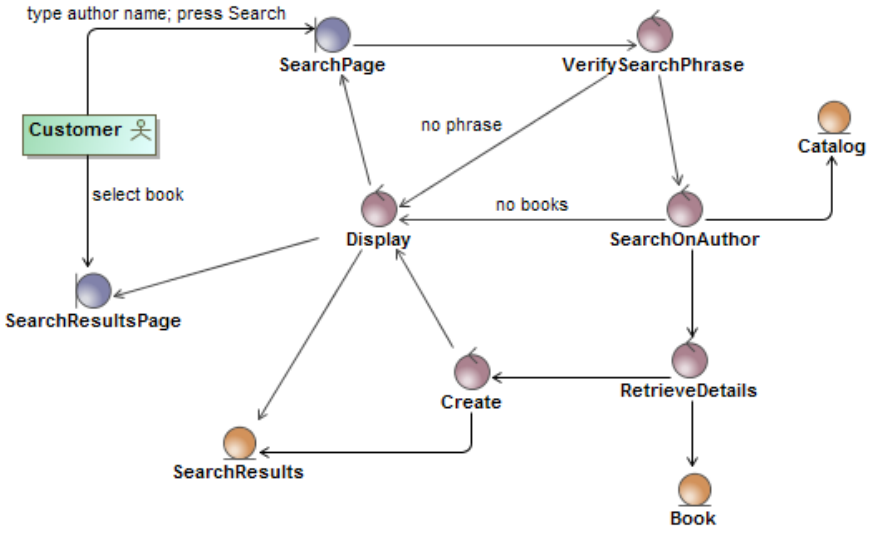
7.3 pav. Dalykinės srities modelis „Internetinė parduotuvė“ 2



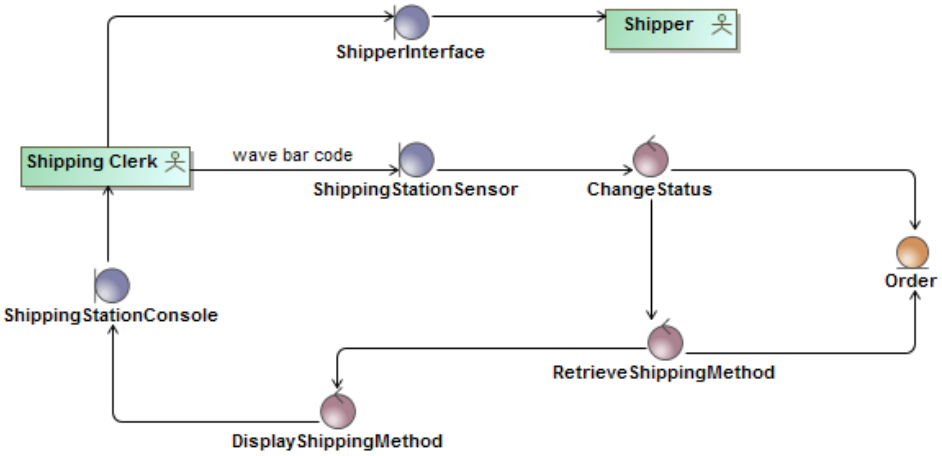
7.4 pav. Išbaigtumo diagrama „Redaguoti pirkinių krepšelį“



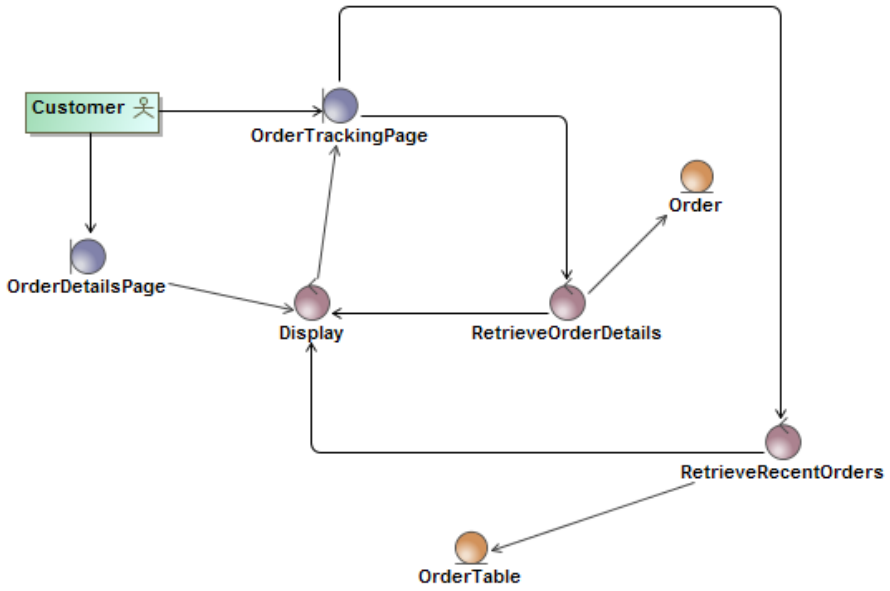
7.5 pav. Išbaigtumo diagrama „Prisijungti“



7.6 pav. Išbaigtumo diagrama „Ieškoti pagal autorių“

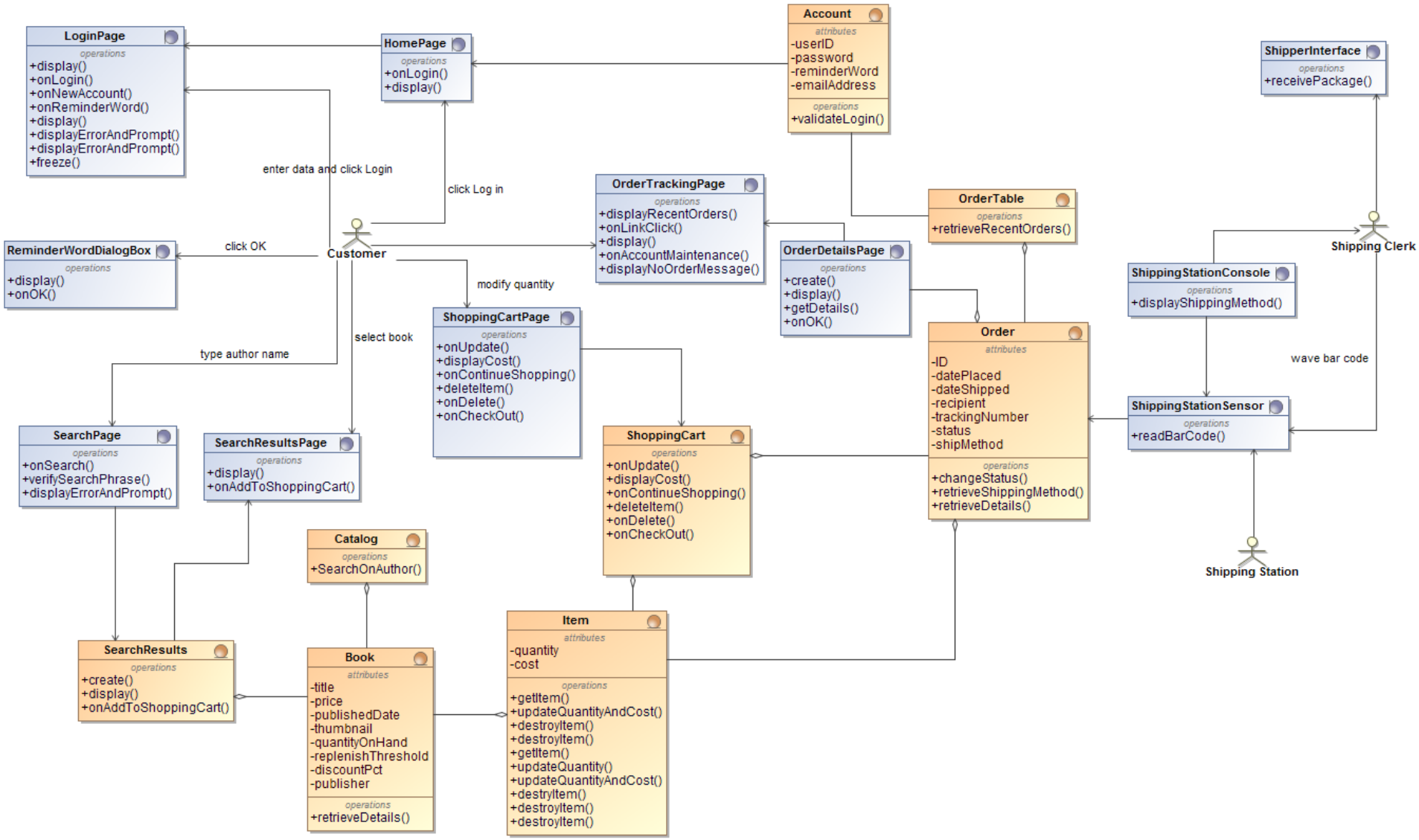


7.7 pav. Išbaigtumo diagrama „Siųsti užsakymą“

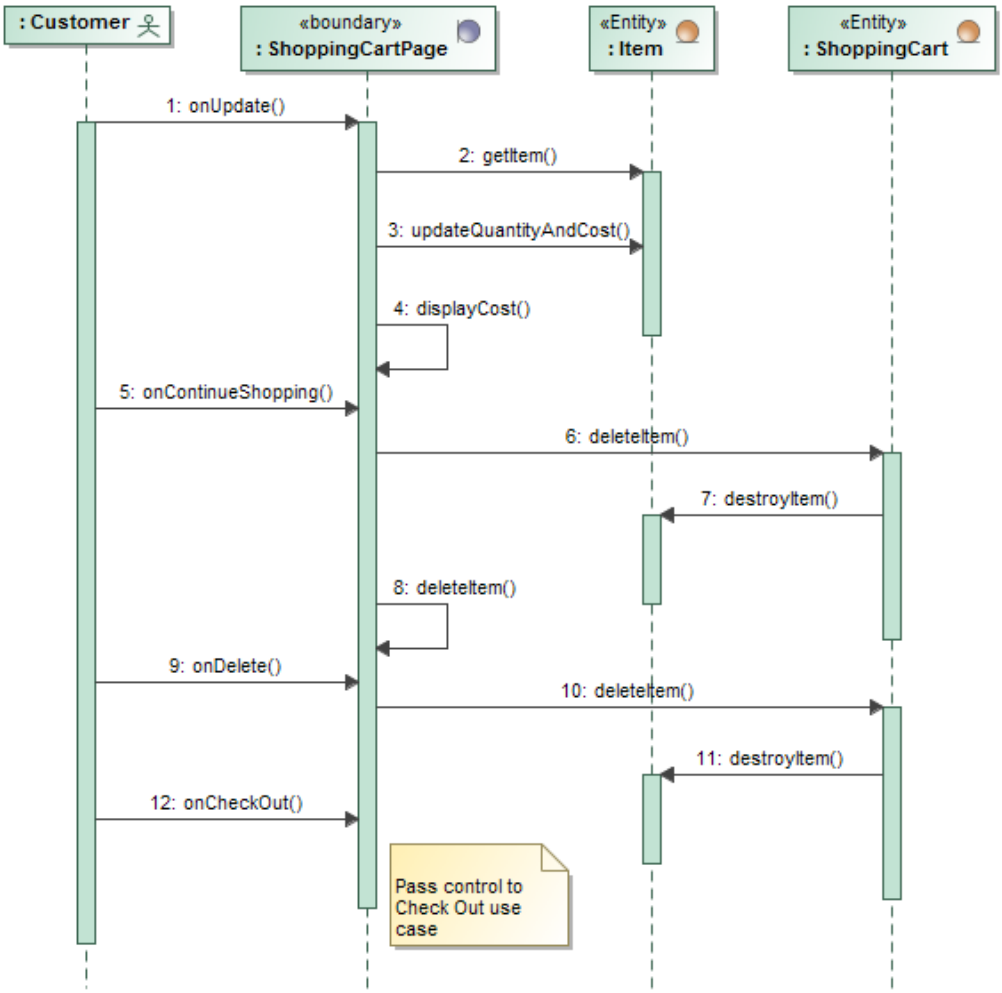


7.8 pav. Išbaigtumo diagrama „Sekti naujausius užsakymus“

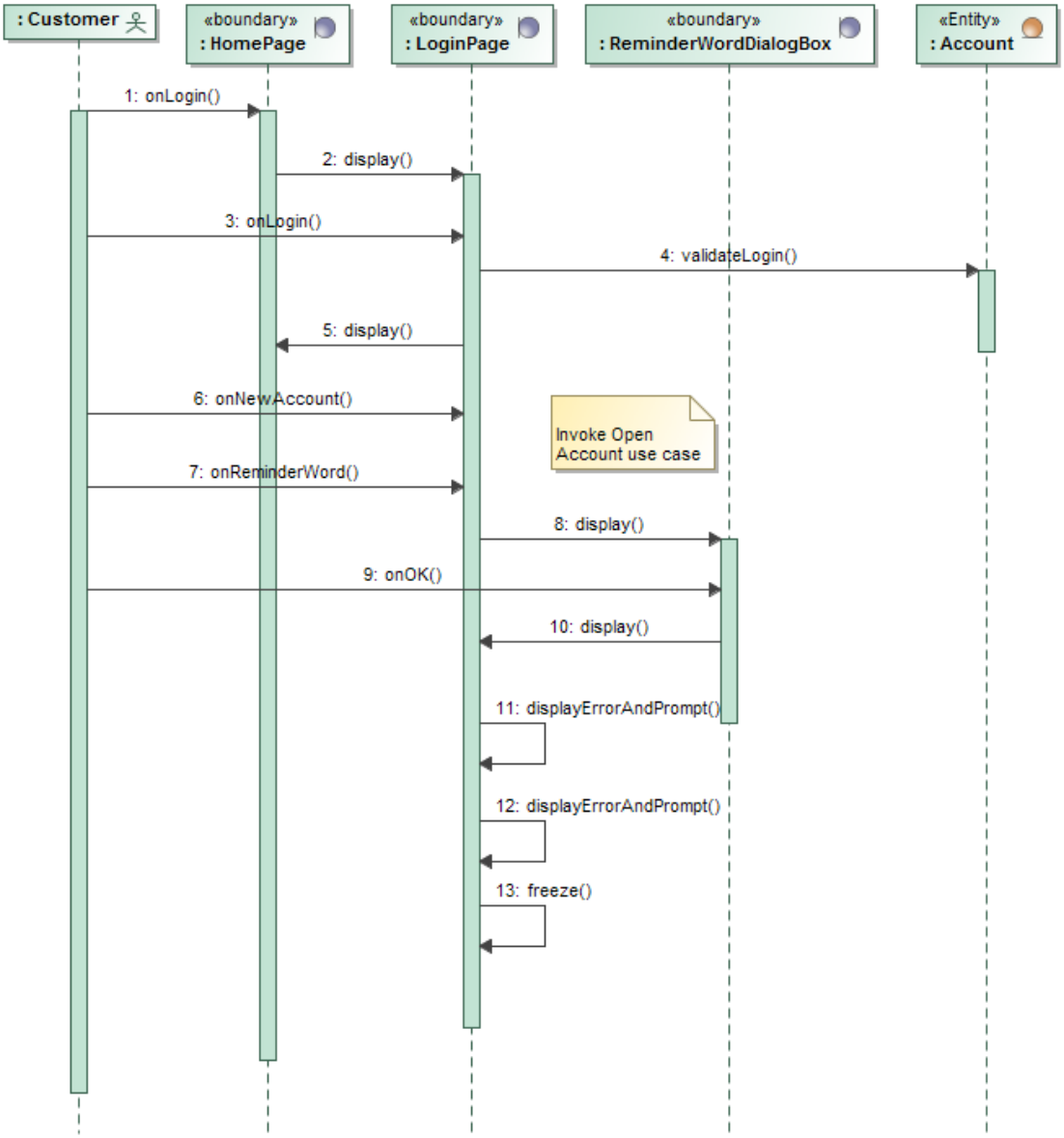
Detalaus projektavimo etapas



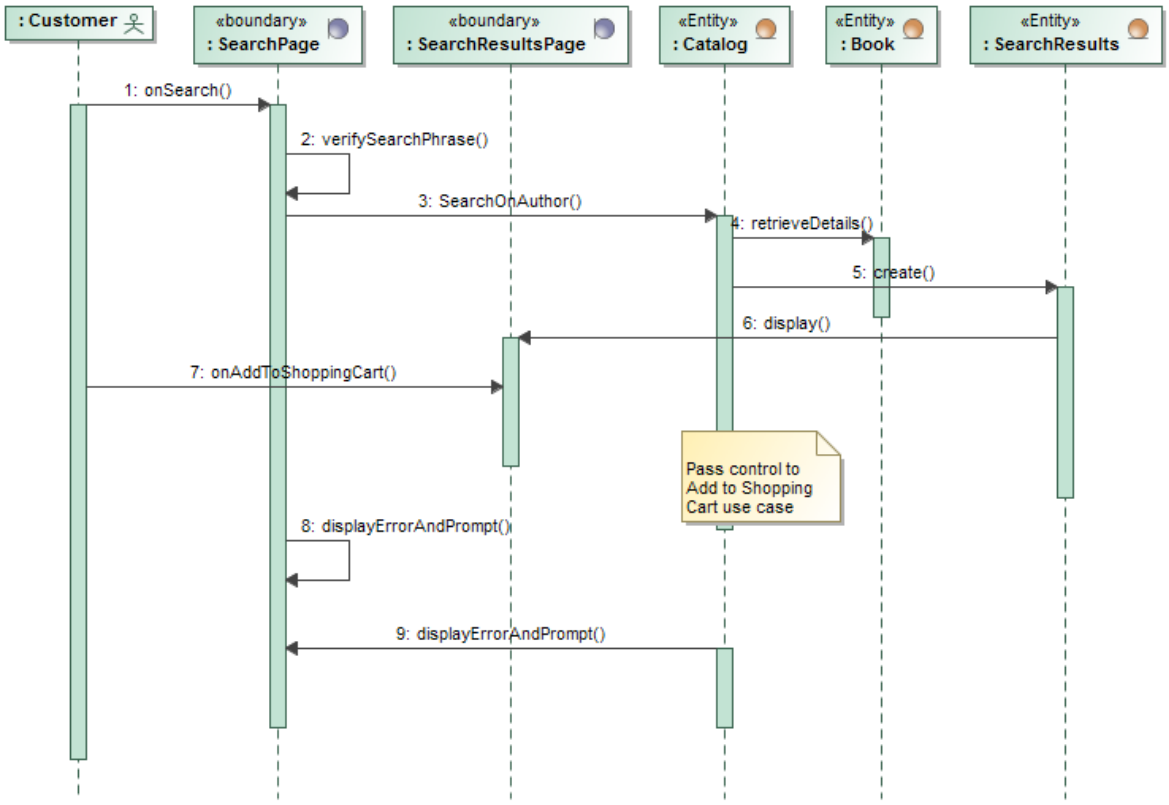
7.9 pav. Statinis modelis „Internetinė parduotuvė“



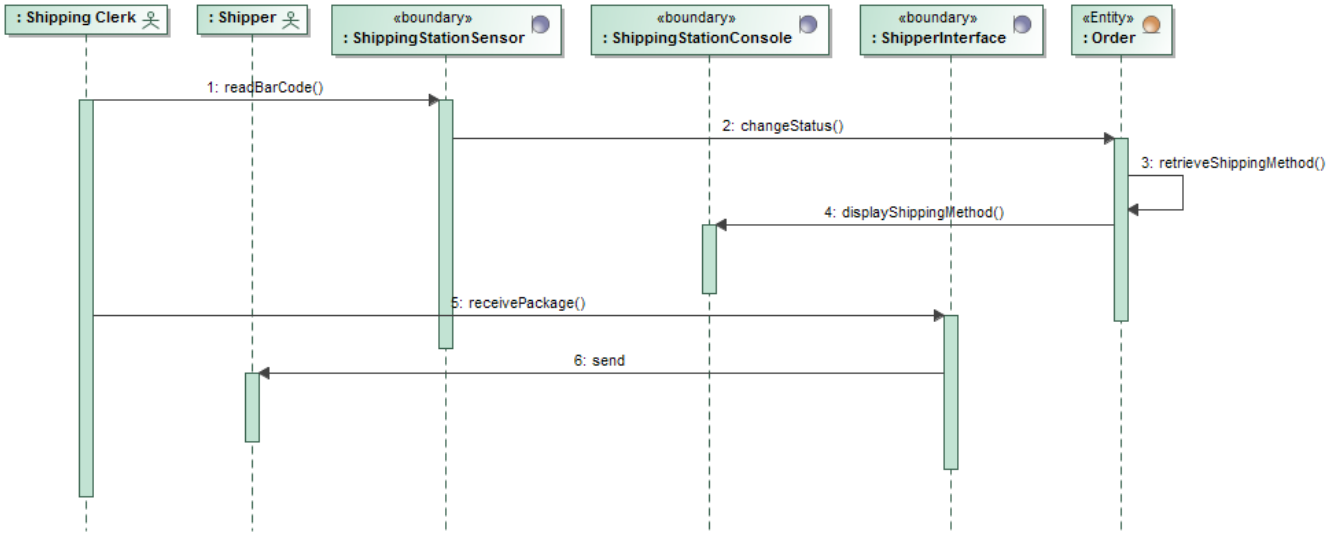
7.10 pav. Sekų diagrama „Redaguoti pirkinių krepšelį“



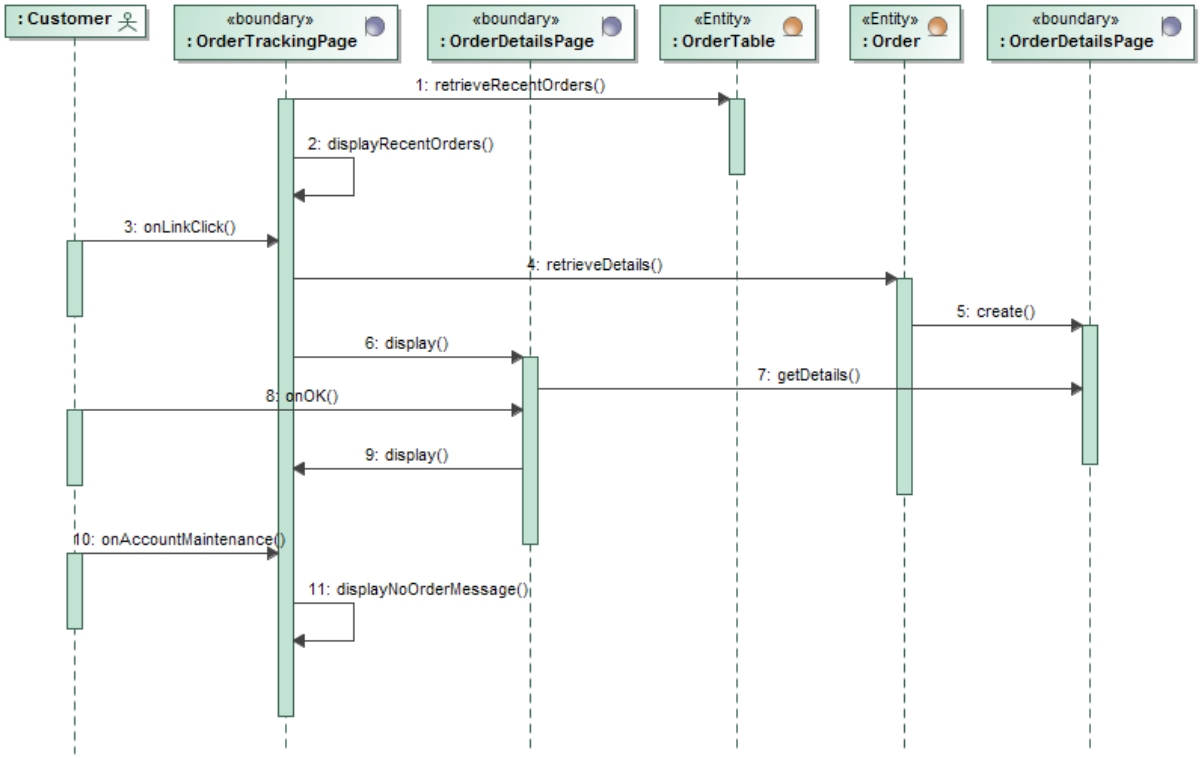
7.11 pav. Sekų diagrama „Prisijungti“



7.12 pav. Sekų diagrama „Ieškoti pagal autorių“



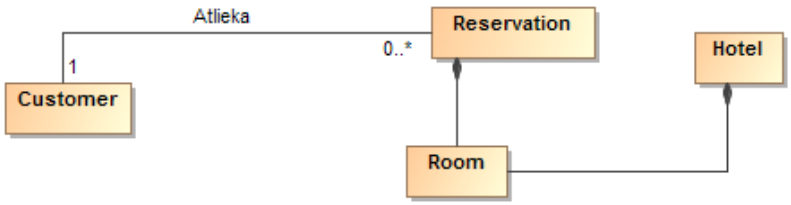
7.13 pav. Sekų diagrama „Siųsti užsakymą“



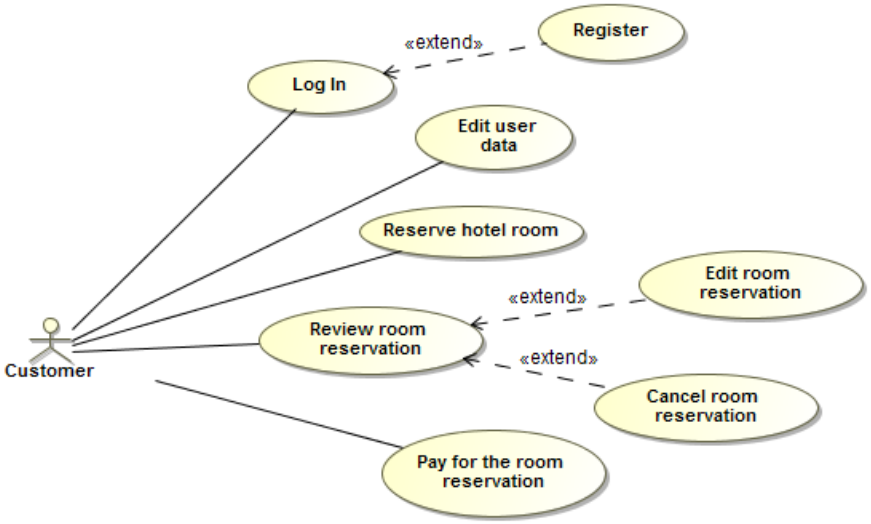
7.14 pav. Sekų diagrama „Sekti naujausius užsakymus“

7.3. Viešbučio kambarių užsakymo IS duomenys

Reikalavimų etapas

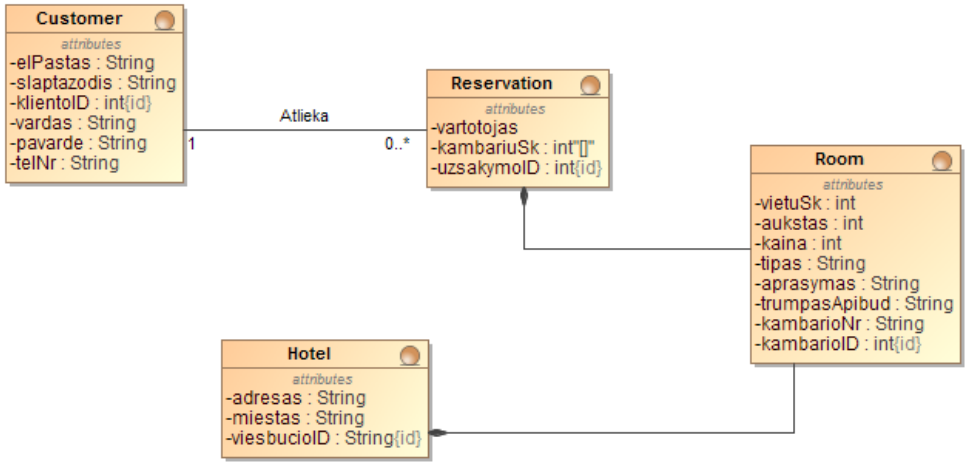


7.15 pav. Dalykinės srities modelis

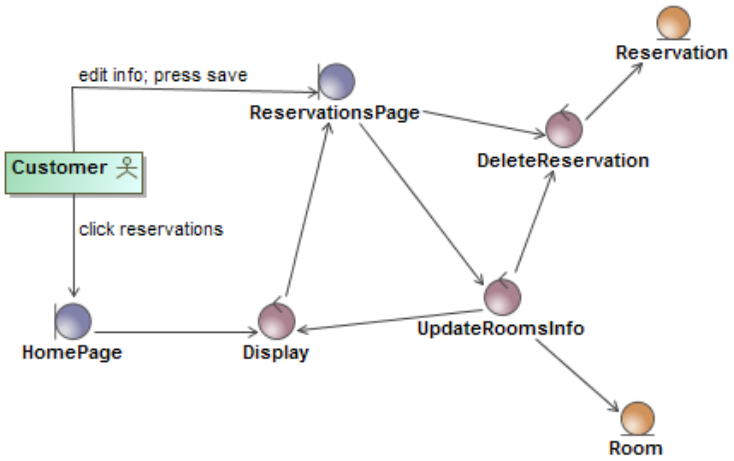


7.16 pav. Panaudojimo atvejų diagrama

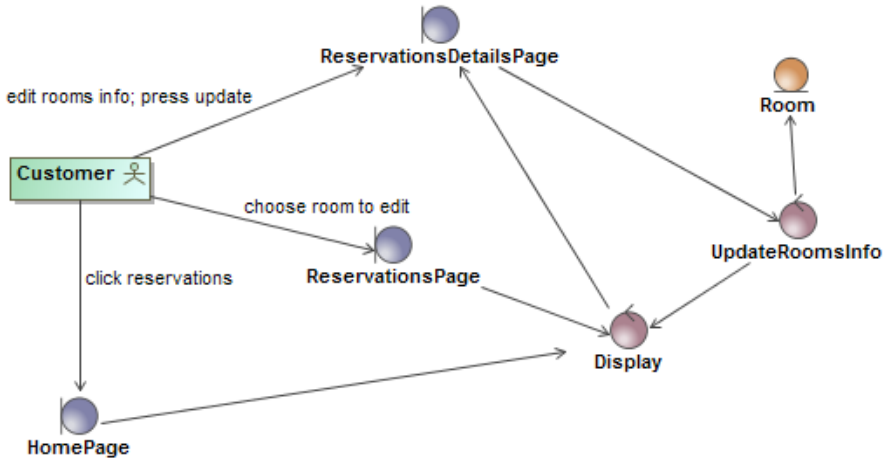
Preliminaraus projektavimo etapas



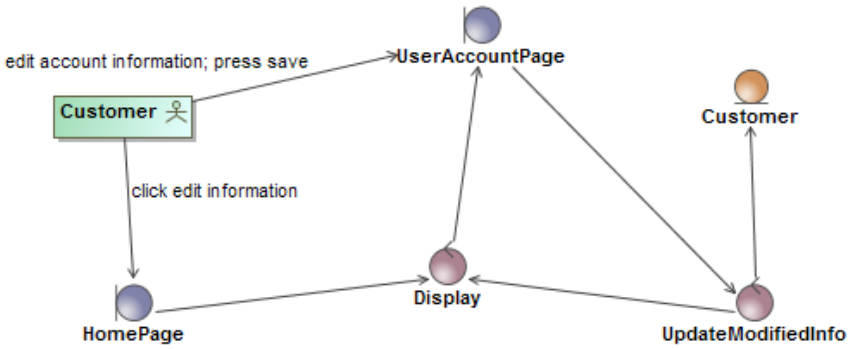
7.17 pav. Dalykinės srities modelis 2



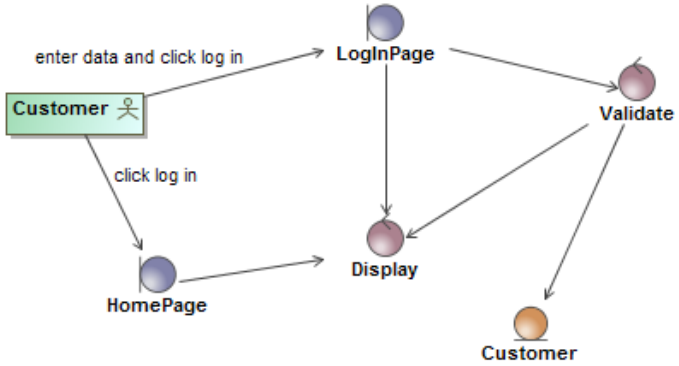
7.18 pav. Išbaigtumo diagrama „Atšaukti kambario rezervaciją“



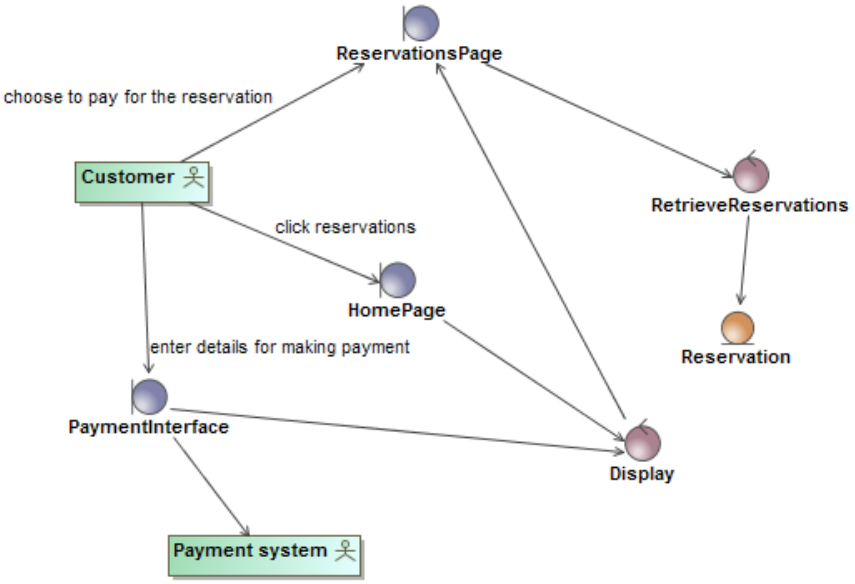
7.19 pav. Išbaigtumo diagrama „Redaguoti kambario rezervaciją“



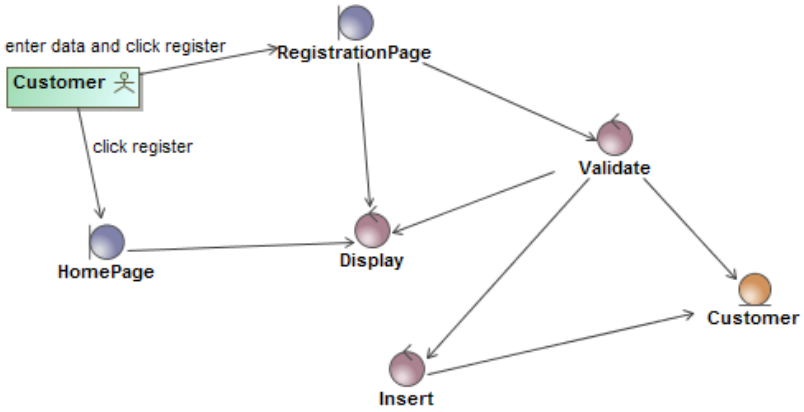
7.20 pav. Išbaigtumo diagrama „Redaguoti vartotojo paskyros duomenis“



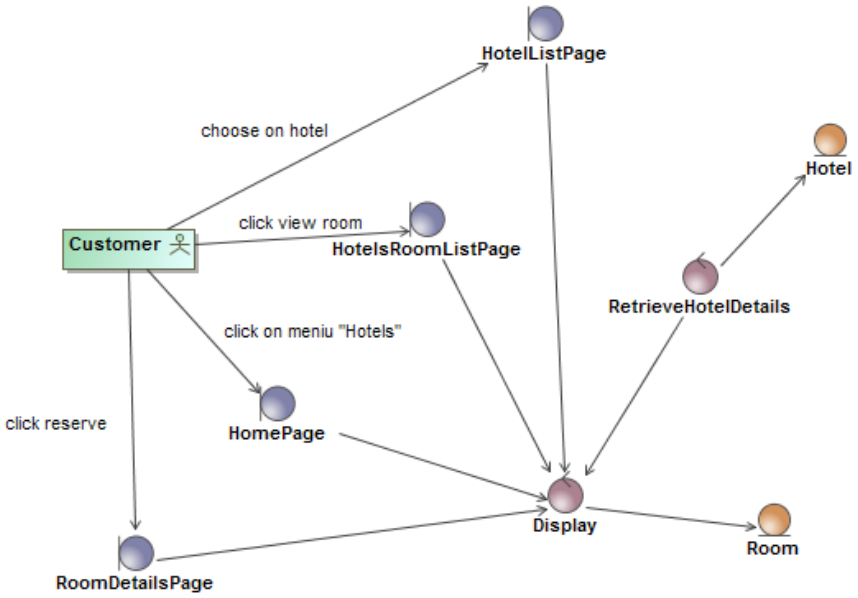
7.21 pav. Išbaigtumo diagrama „Prisijungti“



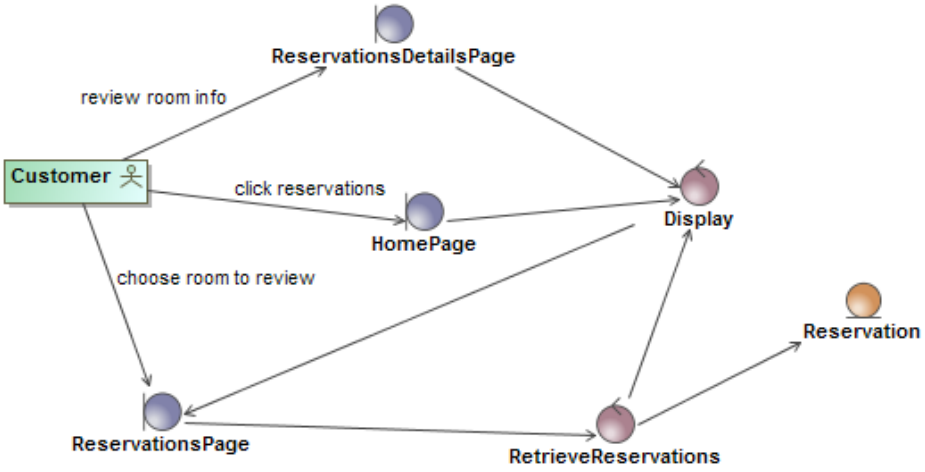
7.22 pav. Išbaigtumo diagrama „Apmokėti kambario rezervaciją“



7.23 pav. Išbaigtumo diagrama „Registruotis sistemoje“

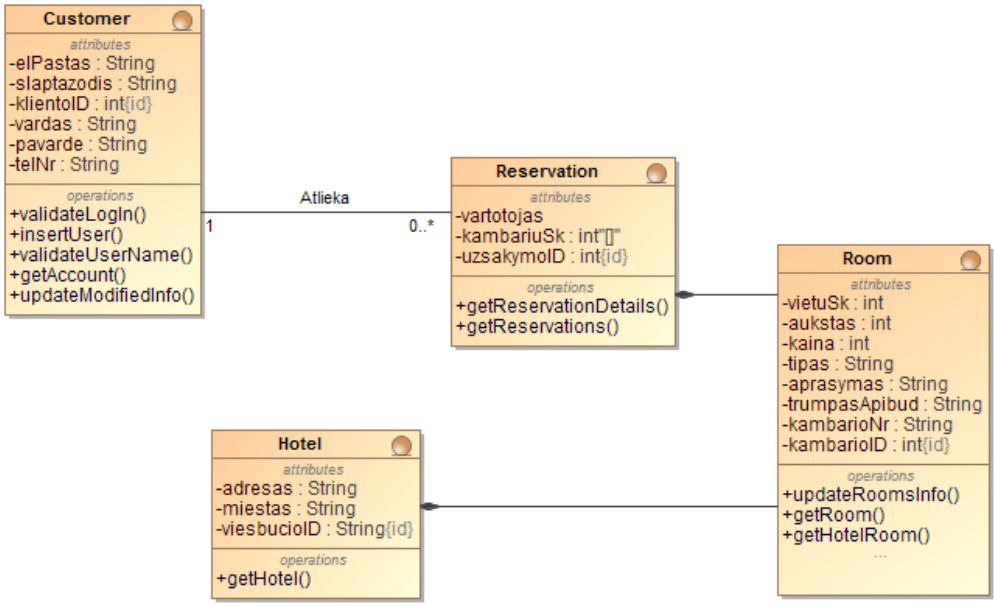


7.24 pav. Išbaigtumo diagrama „Rezervuoti viešbučio kambarį“

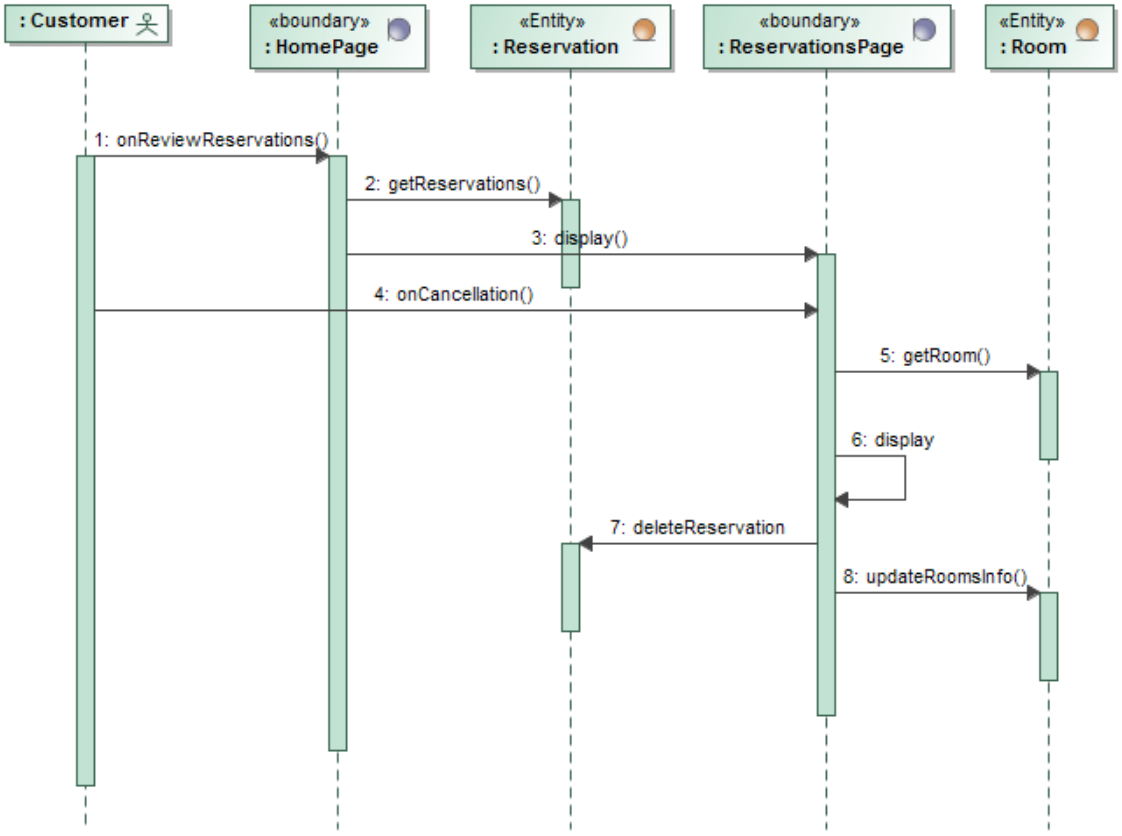


7.25 pav. Išbaigtumo diagrama „Peržiūrėti kambario rezervaciją“

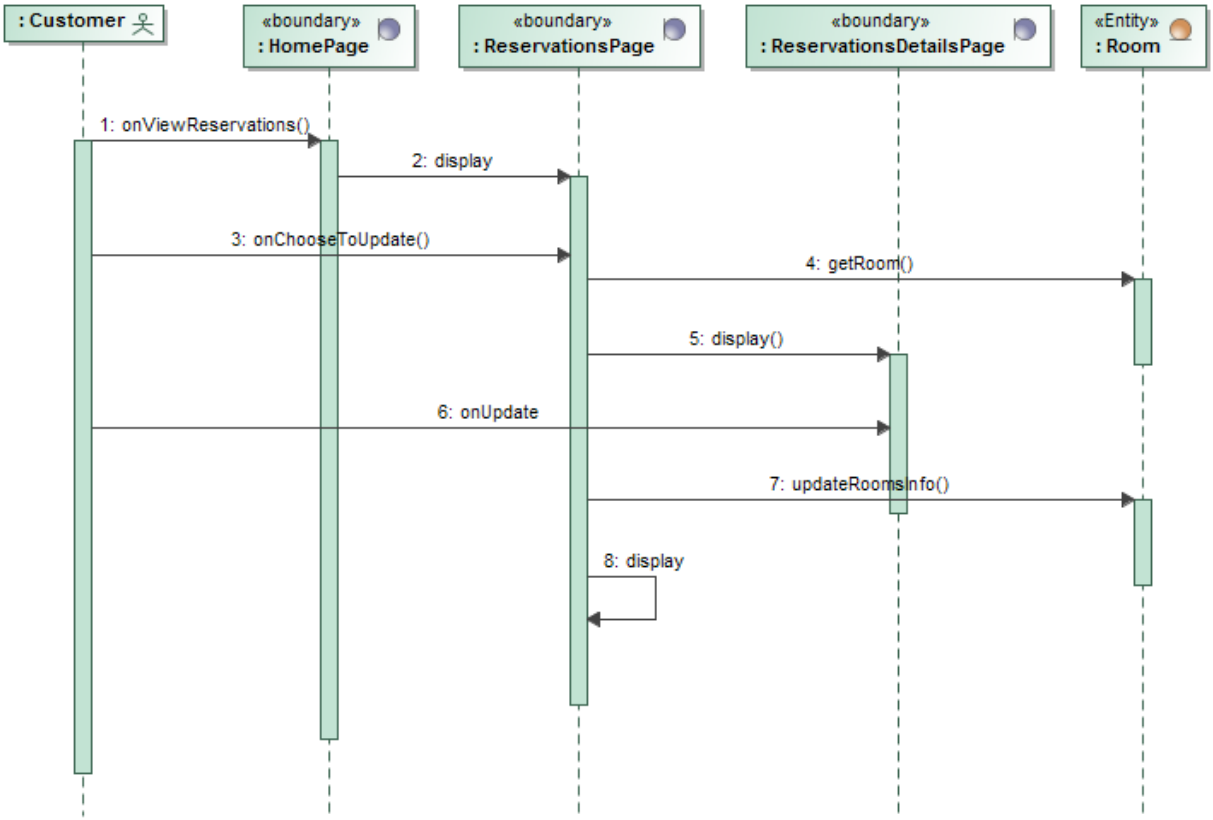
Detalaus projektavimo etapas



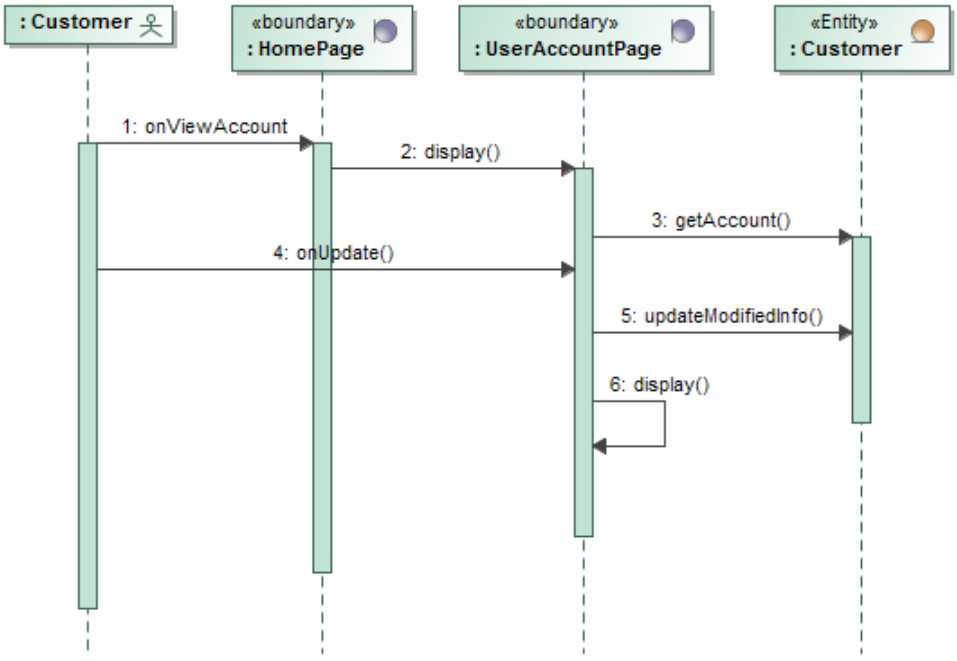
7.26 pav. Dalykinės srities modelis 3



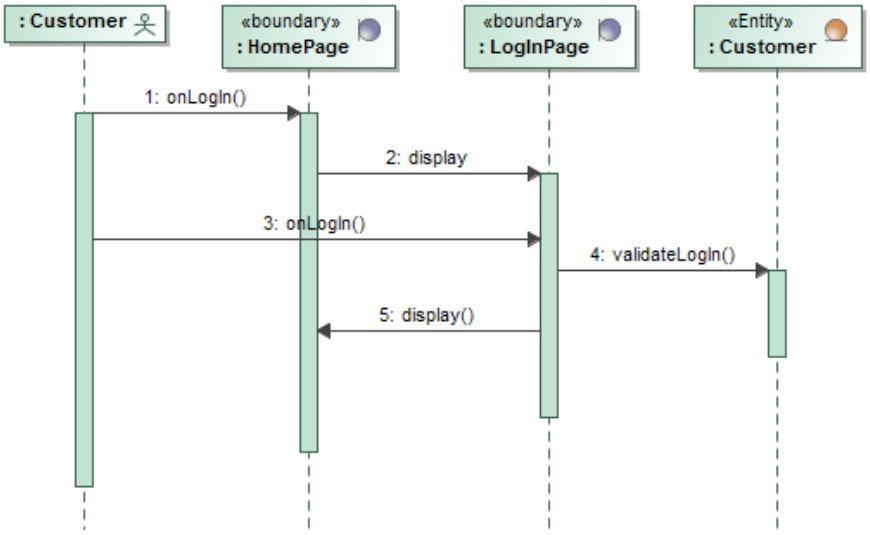
7.27 pav. Sekų diagrama „Atšaukti kambario rezervaciją“



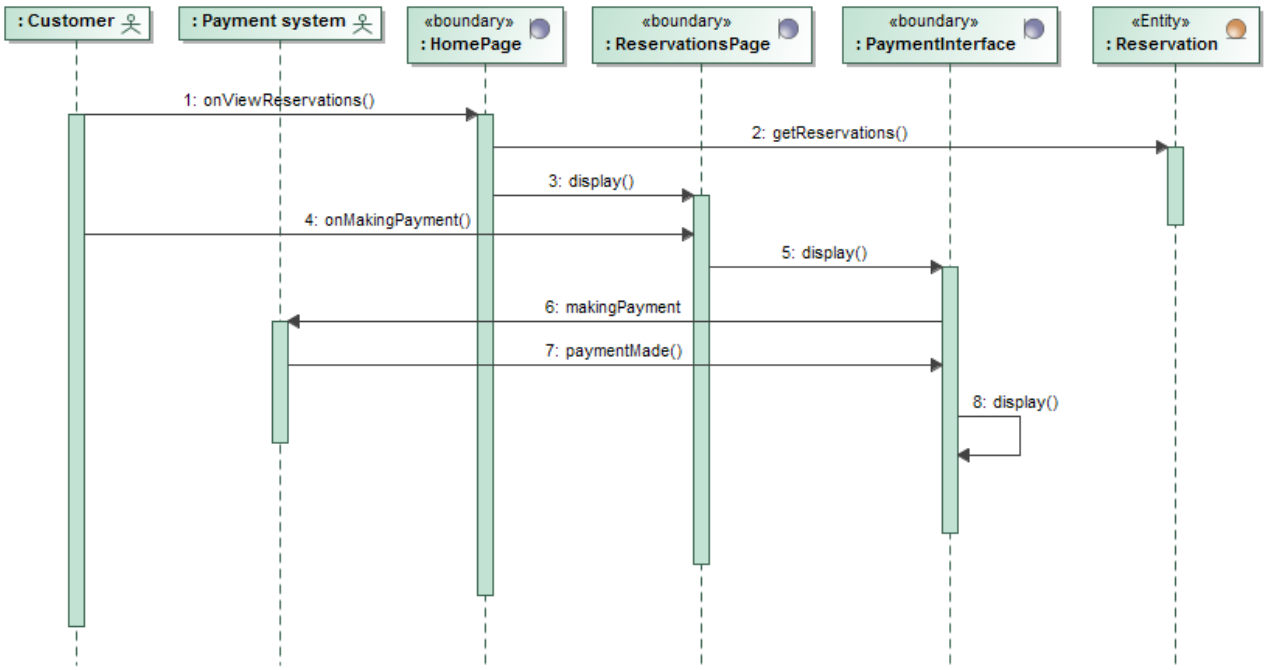
7.28 pav. Sekų diagrama „Redaguoti kambario rezervaciją“



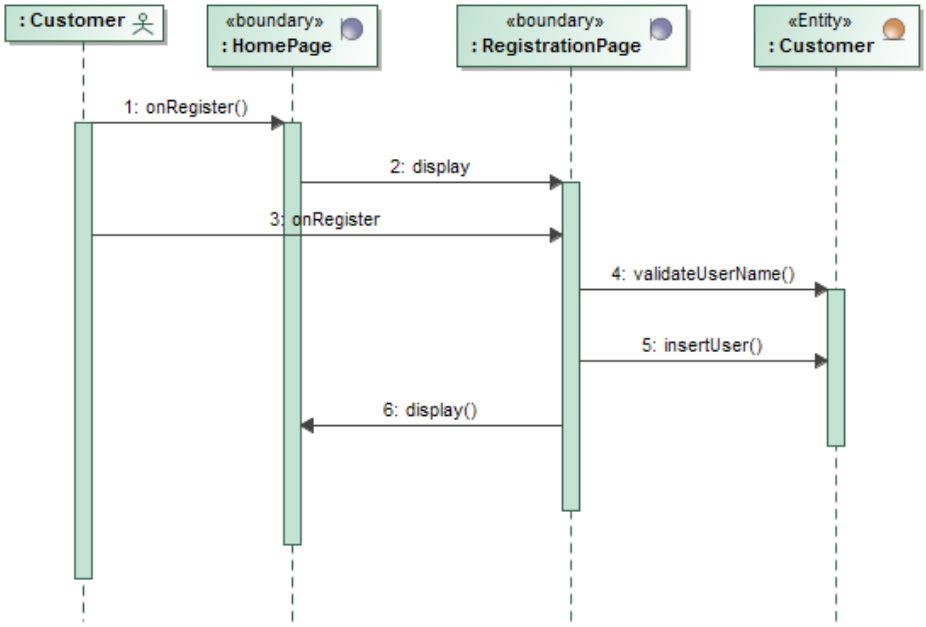
7.29 pav. Sekų diagrama „Redaguoti vartotojo paskyros informaciją“



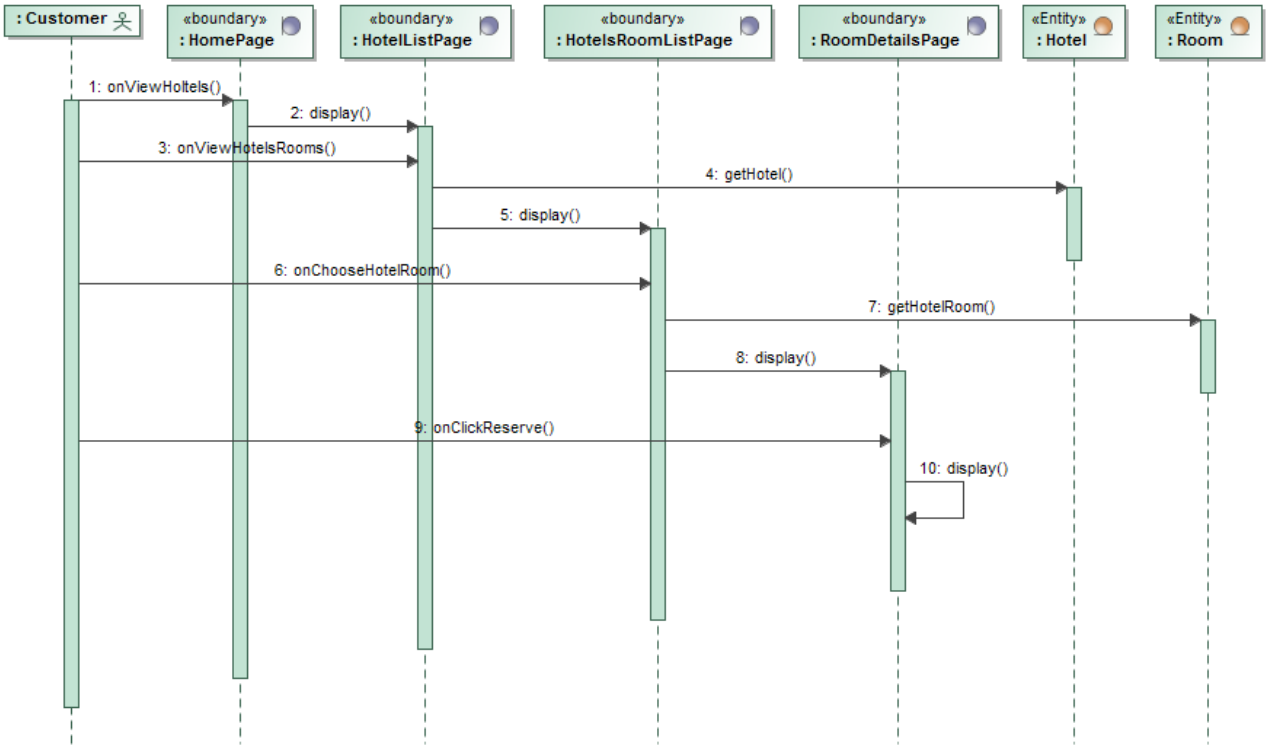
7.30 pav. Sekų diagrama „Prisijungti“



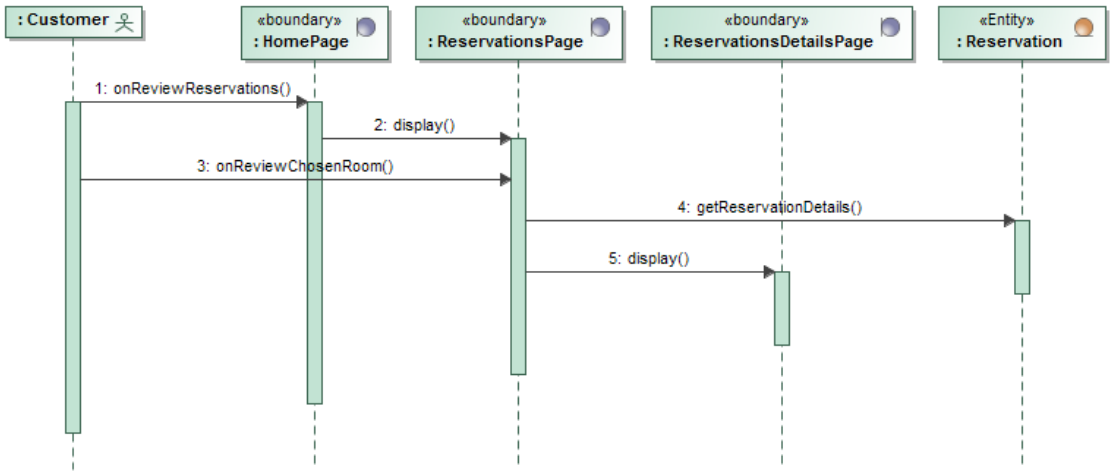
7.31 pav. Sekų diagrama „Apmokėti kambario rezervaciją“



7.32 pav. Sekų diagrama „Registruotis sistemoje“



7.33 pav. Sekų diagrama „Rezervuoti viešbučio kambarį“



7.34 pav. Sekų diagrama „Peržiūrėti kambario rezervaciją