VILNIAUS UNIVERSITETAS

MATEMATIKOS IR INFORMATIKOS FAKULTETAS

PROGRAMŲ SISTEMŲ KATEDRA

Programų sistemų kūrimo metodų praplėtimas rolėmis

Role Extended Software Development Methods

Magistro darbas

Atliko: Jaunius Pisaravičius (parašas)

Darbo vadovas: lekt. Donatas Čiukšys (parašas)

Recenzentas: asist. Karolis Petrauskas (parašas)

Vilnius – 2011

TURINYS

SANTRAUKA

Baigiamajame magistro darbe analizuojami rolinio modeliavimo aspektai įvairių programų sistemų kūrimo metodų ir procesų rėžiuose. Modeliavimo aspektai analizuojami objektinio programavimo evoliucijos kontekste. Apžvelgiamos įvairios prielaidos leidžiančios visuose programų sistemų kūrimo etapuose atskirti sistemos elgseną ir išreikštinai sistemą modeliuoti rolėmis. Darbe identifikuojamos programų sistemų kūrimo procesų dalys, kurios stokoja rolinio modeliavimo instrumentų. Pasiūlomi programų sistemų konstravimo etapo problemų susijusių su elgsenos modeliavimu klasikiniame objektiniame programavime sprendimo būdai. Taip pat pateikiamos programų sistemų kūrimo proceso modifikavimo rekomendacijos dėl rolinio modeliavimo panaudojimo visuose sistemos kūrimo etapuose.

Raktiniai žodžiai: programų sistemų kūrimo procesai ir metodai, rolinis modeliavimas, OOA, OOD, OOP, DCI, į dalykinę sritį orientuotas projektavimas.

SUMMARY

Final master thesis comprises role modeling aspects in various software development processes and methods. The analysis is made in the context of object-oriented programming evolution. The survey is made on miscellaneous assumptions related to explicit role modeling and separation of system state and behaviour concerns in software development. The parts of software development process which have a lack of role modeling techniques are identified in this thesis. The suggestions of solutions to classic object-oriented programming behaviour modeling problems for software construction phase are made. Finally, the recommendations related to role modeling for software development process modification are made to support all the development phases.

Keywords: software development process and methods, role modeling, OOA, OOD, OOP, DCI, domain-driven design.

ĮVADAS

Programų sistemų kūrime plačiausiai naudojama objektinė paradigma. Egzistuoja daug programų sistemų kūrimo procesų, kurie paremti būtent objektine paradigma. Istoriškai vertinant tokius programų sistemų kūrimo procesus, galima suklasifikuoti į dvi grupes: senieji objektiniai procesai ir panaudojimo atvejais (angl. use case) grįsti kūrimo procesai. Senieji objektiniai procesai stokoja gilesnės dalykinės srities analizės ir projektavimo. Dalykinės srities žinios iš karto bandomos konstruoti objektų ir pranešimų (angl. messages) sąvokomis. Šių procesų kertinės sąvokos būtent ir yra objektai ir pranešimai. Tuo tarpu vėliau atsiradę, ir dabar industrijoje plačiai paplitę panaudojimo atvejais grįsti programų sistemų kūrimo metodai įveda daugiau sąvokų bandydami palengvinti, struktūrizuoti sistemų kūrimą, mažinti atotrūkį tarp įvairių suinteresuotų asmenų grupių.

Senuosiuose objektine paradigma grįstuose procesuose praktiškai nėra jokių priemonių kaip būtų galima modeliuoti dalykinės srities elgseną (angl. behaviour) – kompleksines verslo transakcijas, kurių tikslui pasiekti dalyvauja daugiau nei vienas dalykinės srities objektas. Vėlesniuose kūrimo procesuose atsiradusi panaudojimo atvejo sąvoka buvo pirmasis žingsnis į išreikštinį dalykinės srities elgsenos projektavimą. Buvo suprasta, kad būtent sistemos elgsena, o ne struktūra, suteikia pridėtinę vertę. Atsiradus panaudojimo atvejo sąvokai buvo prieita prie kitos svarbios dalykinės srities elgsenos modeliavimo sąvokos – rolės. Rolė yra išreikštinai išskirta dalykinės srities objekto elgsena tam tikrame panaudojimo atvejyje (apibrėžimą suformulavo T. Reenskaug [Ree07]). Pavyzdžiui bankinio pinigų pervedimo atveju bankinės dalykinės srities esybė „sąskaita“ atlieka išreikštinai išskirtas roles: „sąskaita gavėjas“ ir „sąskaita siuntėjas“. Tokiu būdu rolės gali būti trivialiai pakartotinai panaudotos kitų esybių. Taip pat visiškai atskirti lieka programų sistemos struktūros ir elgsenos turiniai. Taigi, buvo suprasta, kad dalykinės srities objektas dalyvaudamas keliuose panaudojimo atvejuose atlieka skirtingas roles panaudojimo atvejo tikslui pasiekti. Dėl rolės sąvokos atsiradimo buvo sukurta aibė rolių modeliavimo metodų. Tačiau tie metodai industrijoje neprigijo, nes persikėlus į programų sistemos konstravimo etapą (iš analizės ir projektavimo etapo) objektinėse programavimo kalbose nebuvo rolės sąvokos kaip pirminio artefakto.

Šiandien jau egzistuoja objektinė programų sistemų kūrimo paradigma pavadinta DCI [RC09] (angl. Data Context Interaction). Būtent DCI paradigma yra ta konstravimo priemonė, kuri pateikia sprendimus, kaip galima konstruoti programinį kodą realizuojant analizės ir projektavimo etapuose

išreikštinai identifikuotus panaudojimo atvejus ir roles. Didžiausią dėmesį DCI skiria dalykinės srities elgsenos – panaudojimo atvejų patogiam ir logiškam realizavimui. Tokiu būdu panaudojimo atvejais grįsti programų sistemų kūrimo metodai įgauna visas priemones reikalingas projektiniams modeliams realizuoti. Tačiau programų sistemos (elgsenos prasme) nėra sudarytos vien tik iš sudėtingų transakcijų, kuriose dalyvauja keletas objektų. Dalykinės srities elgseną taip pat sudaro paprasti, atominiai veiksmai, kurie keičia tik vieno objekto būseną. Tokiai paprastai elgsenai modeliuoti būtų logiška pasitelkti senuosius primityvius objektinius procesus, kurie neleistų atlikti bereikalingus analizės ir projektavimo darbus.

Apibendrinus šiandieninę programų sistemų inžinerijos metodų ir objektinių programavimo kalbų situaciją, galima įžvelgti dvi esmines problemas:

1. Dabartiniai programų sistemų inžinerijos metodai nepateikia konkrečių receptų, kaip programų sistemas būtų galima modeliuoti rolėmis visuose modeliavimo etapuose.
2. Dabartinės objektinės programavimo kalbos ir programų sistemų inžinerijos metodai nepateikia galimybės inkapsuliuoti išreikštinai suprojektuotą dalykinės srities elgseną – panaudojimo atvejus.

Tyrimo aktualumas

Šiandien, programų sistemoms kurti egzistuoja daugybė programų sistemų inžinerijos metodų ir programavimo kalbų. Programų sistemų kūrime plačiausiai naudojama objektinė paradigma. Šiandien plačiai naudojami panaudojimo atvejais grįsti programų sistemų kūrimo metodai apibrėžia rolės sąvoką projektavimo ir analizės etapuose. Tačiau rolės sąvoka kūrimo metoduose nėra pirmarūšė. Egzistuojant programavimo priemonių trūkumui, kurios paremtų apibrėžtą rolės sąvoką, rolės koncepcijos naudojimas tiek analizės, tiek projektavimo etapuose neprigijo. Todėl norint modeliuoti rolėmis visuose programų sistemų kūrimo proceso etapuose būtų verta turėti programavimo priemones, kuriomis būtų galima inkapsuliuoti roles, o tuo pačiu ir panaudojimo atvejus.

Dabartiniai programų sistemų inžinerijos metodai nepateikia konkrečių receptų, kada turėtų būti naudojama elementariosios dalykinės srities elgsenos (atominių operacijų, kurios vykdomos vieno objekto rėmuose) projektavimo metodai, o kada sudėtingų transakcijų įgyvendinimo architektūros – tokios kaip DCI paradigma. Šios problemos išsprendimas stipriai sumažintų šiandien vyraujantį chaosą programų sistemų inžinerijos metodų pasirinkime ir panaudojime.

Darbo tikslas

Keliamas darbo tikslas – parodyti, kaip sprendžiamos įvade įvardintos dvi esminės klasikinio objektinio programavimo problemos panaudojant naują projektavimo, programavimo paradigmą. Pasiūlyti, kaip būtų galima integruoti dalykinės srities elgsenos elementus į programų sistemų kūrimo procesą. Tam yra sukuriamos programų sistemų kūrimo proceso modifikavimo rekomendacijos.

Darbo uždaviniai

Magistro darbo tikslui pasiekti, reikia atlikti šiuos uždavinius:

- Išanalizuoti suformuotas programų sistemų rolinio modeliavimo idėjas.
- Identifikuoti ir išanalizuoti įvarias programų sistemos būsenos ir elgsenos modeliavimo prielaidas.
- Išanalizuoti šiuo metu industrijoje naudojamų programų sistemų kūrimo procesų rolinio modeliavimo panaudojimą.
- Parodyti kaip ir kokias programų sistemų realizavimo problemas sprendžia DCI paradigma.
- Parodyti kaip rolinio modeliavimo panaudojimas visuose kūrimo etapuose modifikuoja objektiškai orientuotų programų sistemų kūrimo procesą.

1. ROLINIS MODELIAVIMAS PROGRAMŲ SISTEMŲ KŪRIME

1.1. Struktūros ir elgsenos modeliavimas

Programų sistemų objektinį modeliavimą galima suskirstyti į dvi fundamentalias aukšto lygio perspektyvas: klasių perspektyvą (struktūros) ir rolių perspektyvą (elgsenos). Klasių perspektyva įtraukia įvairius požiūrius apie programas ir jų dalis tokias kaip klasės, klasių hierarchijos, paketai ir dislokavimas. Tuo tarpu rolių perspektyva apibūdina sąveikaujančius objektus programos vykdymo metu. Bendradarbiavimai (angl. collaborations), panaudojimo atvejai, sąveikos (angl. interactions), būsenų ir veiklos grafai priklauso rolių perspektyvai.

Programų sistemos struktūros modeliavimui rūpi faktai, žinios apie dalykinės srities objektus. Pagrindinis struktūros arba klasių perspektyvos tikslas yra užtikrinti informacijos darną. Tuo tarpu elgsenos modeliavimui rūpi tai, kaip objektai kartu veikia ir pasiekia vieni kitus. Pagrindinis elgsenos modeliavimo tikslas yra užtikrinti teisingą ir maksimaliai efektyvią inkapsuliaciją dėl pakartotinai panaudojamų rolių ir aukštų programų sistemos evoliucijos (prižiūrimumo) savybių.

Kognityvinės psichologijos tyrėjas S. Pinker [Pin97] teigia, kad žmogaus mentaliniame modelyje egzistuoja rolės, kurios kategorizuoja objektus. „Tyrimas parodė, kad žmonės kategorizuoja objektus ne pagal jų esmines charakteristikas, o pagal tai, kokias roles objektai atlieka. ... Artefaktai nėra apibrėžiami pagal jų formą ar struktūrą – tik pagal tai, ką tie artefaktai gali atlikti arba pagal tai, ką kas nors nori, kad jie atliktų.“ Taigi, sistemos komponentus galima kategorizuoti pagal tai, kokią rolę jie atlieka sistemoje. Sistema yra realaus pasaulio dalis, kurią mes pasirenkame laikyti kaip tarpusavyje susijungusių ir sąveikaujančių komponentų visumą. Komponentai sąveikauja, kad pasiektų tam tikrą tikslą – įgyvendintų panaudojimo atvejį. Todėl natūralu, kad programų sistema turėtų būti modeliuojama panaudojimo atvejais (kaip pirmarūšiu modeliavimo įrankiu), o programų sistemos komponentai būtų modeliuojami rolėmis, kurios parodo komponentų tikslus programų sistemoje. Tokios rolės programų sistemoje atitinka artefakto sąvoką S. Pinker psichologiniuose tyrimuose.

Objektinėse programų sistemose rolės sąvoką rolinio modeliavimo vienas iš pradininkų T. Reenskaug savo publikacijoje [Ree07] siūlo apibrėžti kaip dalinį objekto apibrėžimą apibūdinantį objekto atsakomybę ir kitas savybes tam tikrame programų sistemos panaudojimo atvejyje.

Klasių ir rolių perspektyvų sąvokų palyginimas pateiktas 1 lentelėje.

1 lentelė. Modeliavimo perspektyvų sąvokų palyginimas

Klasių perspektyva	„klasė“	„konstravimo laikas“	„struktūra“	„statinis“
Rolių perspektyva	„rolė“	„vykdymo laikas“	„elgsena“	„dinaminis“

Klasių perspektyva ir rolių perspektyva atitinka „kas sistema yra“ (angl. what-the-system-is) ir „ką sistema atlieka“ (angl. what-the-system-does) koncepcijas. Klasėmis projektuojamos dalykinės srities esybės formuoja struktūrinius sistemos elementus – tai, kas sistema yra. Darbe nagrinėjamas į dalykinę sritį orientuotas projektavimas (angl. Domain-Driven Design) kaip idėjų visuma parodanti, kaip galima sukurti ir palaikyti integralią programų sistemos struktūrą. Tuo tarpu rolės – dalykinės srities esybių elgsena yra tai, ką sistema atlieka. Rolinį modeliavimą įgalina keletas rolinio modeliavimo idėjų ir DCI programavimo paradigma.

1.2. Rolės ir klasės

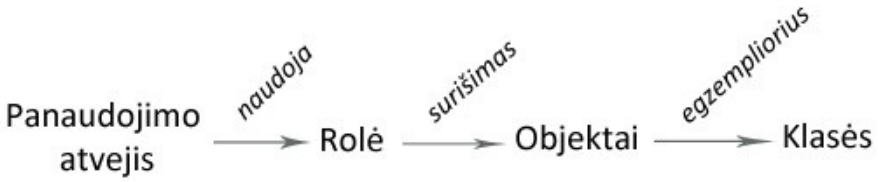
Objektiškai orientuotoje analizėje ir projektavime (OOAD) [Boo93], pirmiausiai, yra aiškiai išskiriamos projektavimo meto (angl. design-time) sąvokos:

- Objektai – tai yra galutinio programų sistemos naudotojo dalykinėje (verslo) srityje egzistuojančios esybės.
- Klasės – jos pateikia paprastą, inkapsuoliuotą prieigą prie dalykinės srities informacijos.

Rolinio modeliavimo idėjos autoriai [MW02, Ree07] prie objekto ir klasės sąvokų prideda ir rolės sąvoką:

- Rolės – konkrečiame programų sistemos panaudojimo atvejyje sąveikaujantys artefaktai su tikslu pasiekti konkretų dalykinės srities (verslo) tikslą.

Paveikslas 1 iliustruoja kaip gali būti organizuota/įgyvendinta programų sistemos elgsena. Panaudojimo atvejis apibūdina kaip programų sistema atlieka užduotį, objektai yra reprezentuojami rolėmis, kurias jie atlieka ir suteikia savo indėlį užduoties įgyvendinimui. Panaudojimo atvejui reikalingi objektai yra vienos ar daugiau klasių egzemplioriai, kurie tuo pačiu yra „tiltas“ tarp užduoties įgyvendinimo (naudojantis rolėmis) ir klasių.



1 pav. Rolės ir klasės sąryšis

1.2.1. Panaudojimo atvejis

Panaudojimo atvejis yra tarpusavyje susijusių rolių struktūra, kuri įgalina programų sistemą atlikti vieną ar daugiau užduočių:

- Panaudojimo atvejis apibūdina bendraujančių rolių struktūrą, kur kiekviena rolė atlieka specifinę funkciją, o visos kartu – įgyvendina reikalingą funkcionalumą.
- Panaudojimo atvejo struktūra gali būti pavaizduota grafu, kur grafo viršūnės yra rolės, o briaunos – rolių sąveikos reprezentacija.

1.2.2. Rolė

Rolės sąvoka atitinka šiuos teiginius:

- Rolė reprezentuoja elgseną.
- Elgsena yra panaudojama (kartu su kitomis rolėmis) panaudojimo atvejo įgyvendinimui.
- Rolė yra įskiepijama į vieną ar daugiau dalykinės srities objektų. Objektai su įskiepyta konkrečia role sakoma, kad atlieka tą rolę.

1.2.3. Objektas

Objektas yra objektiškai orientuoto programavimo sąvoka. Objektas yra tam tikros klasės egzempliorius. Objektas inkapsuliuoja būseną ir elgseną ir turi šias savybes:

- Identifikaciją – savybė atskirianti konkretų objektą nuo kitų sistemos objektų.
- Būseną – objekto atributai ir jų reikšmės.
- Interfeisą – žinučių rinkinį, kurį objektas gali priimti.
- Elgseną – metodus specifikuojančius objekto priimtų žinučių apdorojimą. Elgsena taip pat keičia objekto būseną ir siunčia žinutes tam pačiam arba kitiems objektams.

1.2.4. Klasė

Klasė yra objektų būsenos ir elgsenos specifikacija. Visos sistemos kontekste objekto būsena yra dalis sistemos būsenos, o objekto elgsena – sistemos elgsenos dalis. Klasės sąvoką paremia šie teiginiai:

- Būsena specifikuojama atributų rinkiniu.
- Elgsena specifikuojama metodais, kur kiekvienas metodas yra vykdomas programinis kodas.
- Klasės egzemplioriai (angl. instance) vadinami objektais. Kiekvienas objektas yra unikalus, tačiau kiekvienas turi tas pačias savybes, kurios aprašytos klasėje.

1.3. Praeityje suformuotos rolinio modeliavimo idėjos

Rolėmis grįsto modeliavimo idėjos pirmą kartą buvo pradėtos formuoti Oslo universiteto profesoriaus T. Reenskaug apie 1970 m. Nuo to laiko Oslo universitetas buvo pagrindis rolinio modeliavimo idėjos metodų ir įrankių vystytojas. 1996 m. T. Reenskaug išleido knygą apie objektiškai orientuotų rolių analizę ir modeliavimą (angl. Object Oriented Analysis and Modeling, OOram). Tokiu būdu OOram tapo UML notacijos pirmtaku ir paveldėjo rolinio modeliavimo įrankį – bendradarbiavimo diagramą (angl. collaboration).

Apie 1990 m. R. Wirfs-Brock suformavo atsakomybėmis grįsto projektavimo (ang. Responsibility-Driven Design) idėją. Idėja pastūmėjo galvojimą apie objektus kaip apie duomenis ir algoritmus į objektus kaip roles ir atsakomybes. Atsakomybėmis grįstas projektavimas tapo praktiškai visų objektinių projektavimo idėjų pirmtaku: testais (angl. test-driven), elgsena (angl. behaviour-driven), dalykine sritimi (angl. domain-driven) grįsto projektavimo (išskyrus duomenim grįstą projektavimą).

Nepaisant anksti suprastos objektų rolių svarbos programų sistemų kūrime, rolinis modeliavimas programų sistemų kūrimo rinkoje netapo populiariu modeliavimo metodu dėl tos priežasties, kad trūko tiek teorinės bazės, tiek programavimo kalbų priemonių, kaip analizės ir projektavimo etapuose sumodeliuotos rolės galėtų būti „perkeltos“ į programinį kodą. Šiandien tas pats rolinio modeliavimo autorius T. Reenskaug pristato paradigmą leidžiančią patogiai panaudoti sumodeliuotas roles objektinėse programavimo kalbose.

2. PS BŪSENOS IR ELGSENOS MODELIAVIMĄ ĮGALINANČIOS PRIELAIDOS

2.1. Praktikoje naudojami programų sistemų kūrimo procesai

Programų sistemų kūrimo procese analizuojant dalykinę sritį (diskutuojant su dalykinės srities ekspertais) yra sukuriamas dalykinės srities modelis. Dalykinės srities modelyje identifikuojama dalykinės srities struktūra ir elgsena. Vėlesnėse programų sistemų kūrimo proceso stadijose – sistemos projektavime ir konstravime, kuriamas programų sistemos modelis dalykinės srities modeliu.

Vienas pirmųjų programų sistemų gyvavimo ciklo modelis, kuris remiasi transformaciniu požiūriu yra konstruktyvusis programų sistemų gyvavimo ciklo modelis [Čap96], (angl. Constructive Software Lifecycle Model). Modelyje numatytos dvi šakos - apibendrinimo arba abstrakcijos ir konkretizavimo arba reifikacijos. Todėl įrodomasis programų sistemos gyvavimo ciklo modelis literatūroje dažnai yra vadinamas "dviejų kojų" modeliu. Nuo to laiko, kai buvo suformuluotos pagrindinės modelio idėjos praėjo virš trisdešimt metų.

Pagal „dviejų kojų“ modelį dalykinės srities sąvokos transformuojamos perkėlimo iš dalykinės srities modeliavimo į programų sistemos modeliavimą pasekoje. Transformacinis modelio modifikavimas šiuo metu yra vyraujantis programų sistemų kūrimo būdas. Daugelyje rinkoj populiarių programų sistemų kūrimo procesų remiamasi konstruktyviojo modelio principais – dalykinės srities modelio transformacija į programų sistemos modelį(-ius). To pasekmė – skirtingi modeliai.

Praktikoje naudojama populiari Unifikuotų procesų (UP, angl. Unified Process) šeima. UP sudaro tokios realizacijos kaip RUP [Rat98] (angl. Rational Unified Process), OpenUP [Ecl94] ir keletas kitų procesų. Šie programų sistemų kūrimo procesai atskiria dalykinės srities modeliavimą (angl. Business Modeling) nuo programų sistemos projektavimo ir architektūros modeliavimo. RUP ir OpenUP kūrimo procesai rekomenduoja turėti atskirus modelius dalykiniai sričiai ir programų sistemai. O taip pat kūrimo procesai pateikia nurodymus, kaip efektyviai vykdyti trasavimą iš dalykinės srities modelių į programų sistemos modelius. Tai parodo, kad procesai remiasi transformaciniu modeliavimo principu.

Programų sistemų kūrimo procesas Comet [COM07] griežtai išskiria dvi modeliavimo sritis – dalykinę ir sisteminę. Kūrimo procesas teigia, kad modelis esantis dalykinėje srityje niekada nebus

vienodas modeliui esančiam sisteminėje srityje. Sisteminėje srityje būtinai atsiras papildomų apribojimų nesančių realiame pasaulyje, kurie modifikuos modelį, pridės papildomų infrastruktūrinių aspektų. Iš esmės tai yra transformaciniai modeliavimo žingsniai.

Apibendrinant galima teigti, kad jeigu programų sistemų kūrimo procesas savo rekomendacijose išskiria dalykinės srities modelį (angl. business model) kaip skirtingą modelį programų sistemos modeliui, tai reiškia, jog kūrimo procesas remiasi transformaciniu požiūriu.

Šiandien yra atsiradę prielaidų transformacinį modeliavimo požiūrį palaipsniui keisti evoliuciniu (angl. elaboration). Prielaidas tam sudaro: į dalykinę sritį orientuotas projektavimas (DDD), naujos paradigmos (DCI), technologinių platformų galimybės (Java Enterprise Edition, Java EE). Toliau darbe bus nagrinėjami įvairūs dalykinės srities ir programų sistemų modeliavimo būdai, kurie padeda dalykinės srities modelio evoliucijai (plėtojimui, angl. elaboration) vykdyti.

2.2. Į dalykinę sritį orientuota analizė ir projektavimas

Programų sistemų kūrimo bendruomenė pripažįsta, kad dalykinės srities (angl. domain) modeliavimas yra svarbus programų sistemų kūrimo aspektas. Dėl dalykinės srities projektavimo, programuotojai gali išreikšti sudėtingą funkcionalumą ir jį įgyvendinti sukurdami vykdomą programų sistemą, kuri tenkina vartotojų poreikius. Nepaisant akivaizdžios dalykinės srities modeliavimo svarbos, egzistuoja labai mažai praktinių patarimų kaip į programų sistemų kūrimo procesą efektyviai įjungti dalykinės srities modeliavimą.

Į dalykinę sritį orientuotas projektavimas užpildo tą spragą pateikdamas sistemišką požiūrį į dalykinės srities projektavimą, kartu su rinkiniu geriausių projektavimo praktikų, patirtimi grįstų technikų ir fundamentalių principų, kurie palengvina kompleksiškos dalykinės srities programų sistemų kūrimą.

Pirmiausia, norint sukurti suinteresuotų asmenų interesus tenkinančią programų sistemą reikia žinoti ir suprasti dalykinę sritį, kuriai sistema yra kuriama. Dalykinę sritį geriausiai išmano dalykinės srities ekspertai – žmonės dirbantys tame versle, kuriam kuriama programų sistema. Programų sistemos kuriamos tam, kad patobulintų tam tikros dalykinės srities procesus. Dalykinę sritį galima pavadinti programų sistemos širdimi.

Analizuojant ir kaupiant informaciją apie dalykinę sritį iš dalykinės srities ekspertų gaunama informacija, kuri dar negali būti lengvai panaudojama kaip programų sistemos konstrukcijos. Pirmiausia iš sukauptų žinių turėtų būti sukurta abstrakcija – dalykinės srities modelis. Modelis yra esminė programų sistemos projektavimo dalis. Jis padeda lengviau susitvarkyti su kompleksiškumu.

Dalykinės srities modelis turi būti gerai išdiskutuotas ir suprastas visų suinteresuotų asmenų. Kad tai būtų įmanoma pasiekti reikalingas išreikštinai sukurtas dalykinės srities žodynas – bendroji kalba (angl. ubiquitous language). Bendroji kalba padeda vienareikšmiškai, pilnai, tiksliai išdiskutuoti ir suprasti dalykinės srities modelį.

DDD programų sistemų projektavimą ir konstravimą išlaiko kaip labai tampriai susietas praktikas. Parodo, kad tamprus projektavimas ir konstravimas geba sukurti geresnius sprendimus. Grįžtamasis ryšys tarp šių veiklų yra labai svarbus – geras programų sistemos projektas padės geriau ją sukonstruoti, tuo tarpu grįžtamasis ryšys iš konstravimo proceso pagerins programų sistemos projektą. Tokį tamprų ryšį tarp programų sistemos kūrimo veiklų padeda įgyvendinti DDD rekomenduojamos naudoti judriosios (angl. agile) kūrimo metodologijos (eXtreme Programming, Scrum). Dalykinės srities modelio integralumą, išeities kodų lankstumą ir skaitomumą padeda įgyvendinti aibė DDD rekomenduojamų detalaus ir architektūrinio projektavimo šablonų.

2.2.1. Dalykinės srities esybės ir objektai-reikšmės

Esminė dalykinės srities esybę apibrėžianti charakteristika yra unikalus identifikatorius. Šis identifikatorius yra unikalus visam programų sistemos kontekste. Jokia kita esybė, nesvarbu kiek ji panaši, negali turėti tokio paties unikalaus identifikatoriaus. Kitu atveju tai gali privesti prie prieštaringų duomenų sistemoje. Esybės unikalus identifikatorius gali būti įvairiai realizuotas: unikalus skaitinis identifikatorius, unikalus kodas (angl. Guid – General unique identifier) arba natūraliai iš dalykinės srities atkeliavęs unikalus numeris. Standartiniai esybių pavyzdžiai gali būti tokie: klientas, produktas, kontaktinis asmuo ir t.t.

Esminė objektą-reikšmę apibūdinanti charakteristika yra unikalumo nebuvimas. Objektas-reikšmė, kitaip nei esybė, neturi savo unikalaus identifikatoriaus. Objekto-reikšmės tikslas yra atstovauti kažkokį dalykinės srities objektą tik per savo atributų reikšmes. Tai yra, du objektai-reikšmės gali turėti identiškus atributus ir tokiu atveju tie objektai bus identiški. Programų sistemai jie neturi kitos reikšmės apart savo atributų reikšmių. Standartiniai objektų-reikšmių pavyzdžiai yra: valiuta, miestas, produkto kodas ir t.t.

2.2.2. Dalykinės srities servisai

Kai yra analizuojama dalykinė sritis ir bandoma identifikuoti pagrindinius objektus, kurie sudarys dalykinės srities modelį, kartais aptinkama, kad kai kurie dalykinės srities aspektai negali būti trivialiai priskirti kažkuriai

dalykinės srities esybei. Bendrąja prasme esybės suprantamos kaip objektai turintys atributus (t.y. vidinę savo būseną) ir primityvią elgseną. Kai kuriama ir analizuojama dalykinės srities kalba (terminologija), į ją įtraukiami esminiai dalykinės srities aspektai. Tuomet kalboje esantys reikšminiai daiktavardžiai dažniausiai tampa dalykinės srities modelio esybėmis, o su daiktavardžiais susieti veiksmažodžiai apibrėžiami kaip tų esybių elgsena. Tačiau dažniausiai identifikuojami ir tokie veiksmažodžiai, kurie nėra priklausomi jokioms esybėms arba perkertantys daug esybių. Tokie veiksmai dažnai būna svarbiausi dalykinės srities elementai. DDD tokie veiksmai patalpinami į atskirus elementus, kurie neturi vidinės būsenos. Jie tiesiog teikia funkcionalumą dalykinei sričiai. Tokie objektai vadinami servisais. Pavyzdžiui, servisu galėtų tapti lėšų persiuntimo iš vienos banko sąskaitos į kitą funkcionalumas. Tokia operacija „persiųsti lėšas “, aišku, yra svarbi dalykinės srities operacija. Ji jokiu būdu negali būti ignoruota, tačiau jos pridėjimas kažkokiai dalykinės srities esybei būtų nekorektiškas.

2.2.3. Saugyklos, agregatai, gamyklos

Saugyklos projektavimo šablonas DDD atveju atskiria dalykinės srities objektus (juos palikdamas „švarius“) nuo įvairaus infrastruktūrinio kodo, skirto gauti arba talpinti duomenis į duomenų bazes. Šio šablono panaudojimas leidžia dalykinės srities objektams nesirūpinti jų saugojimu duomenų bazėse, taip leisdamas labiau koncentruotis į dalykinę sritį, o ne į technines detales.

Agregato projektavimo šablonas apima tas esybes ar objektus-reikšmes, kurie dalykinės srities požiūriu turi kažką bendro. Agregato šakninis elementas yra esybė, su išoriniais objektais jungianti visas esybes agregatui priklausančias esybes ar objektus-reikšmes. Kai kurių objektų buvimas sistemoje neturi jokios prasmės be jų tėvinių objektų. Pavyzdžiui, objektas „adresas“. Be priklausomybės klientui toks objektas yra beprasmis. Jis būtų priskirtas agregatui, kurio šakninis elementas būtų „klientas“. Agregatas prisiima atsakomybę už visų jo elementų korektišką panaudojimą. Tokiu būdu išoriniai (agregato požiūriu) objektai lieka „švarūs“ nuo jiems nereikalingos logikos, matydami tik sąveikaudami tik su agregato šakniniu elementu.

Esybės ir agregatai kartais gali būti dideli ir kompleksiški, jų sukūrimas gali reikalauti daug žinių apie jų vidinę sandarą. Tokiais atvejais į pasitelkiamas projektavimo šablonas – gamykla. Gamykla inkapsuliuoja visas žinias reikalingas objektams sukurti. Tai ypač naudinga agregatų sukūrimui. Agregato gamykla turi visas žinias apie agregato šakninio elementų ir visų kitų elementų sukūrimo logiką. Tai vėlgi leidžia objektams, kurie naudojasi tuo agregatu atsiriboti nuo jiems nepriklausančios dalykinės srities žinių.

2.2.4. Daugelio lygmenų architektūrinis stilius

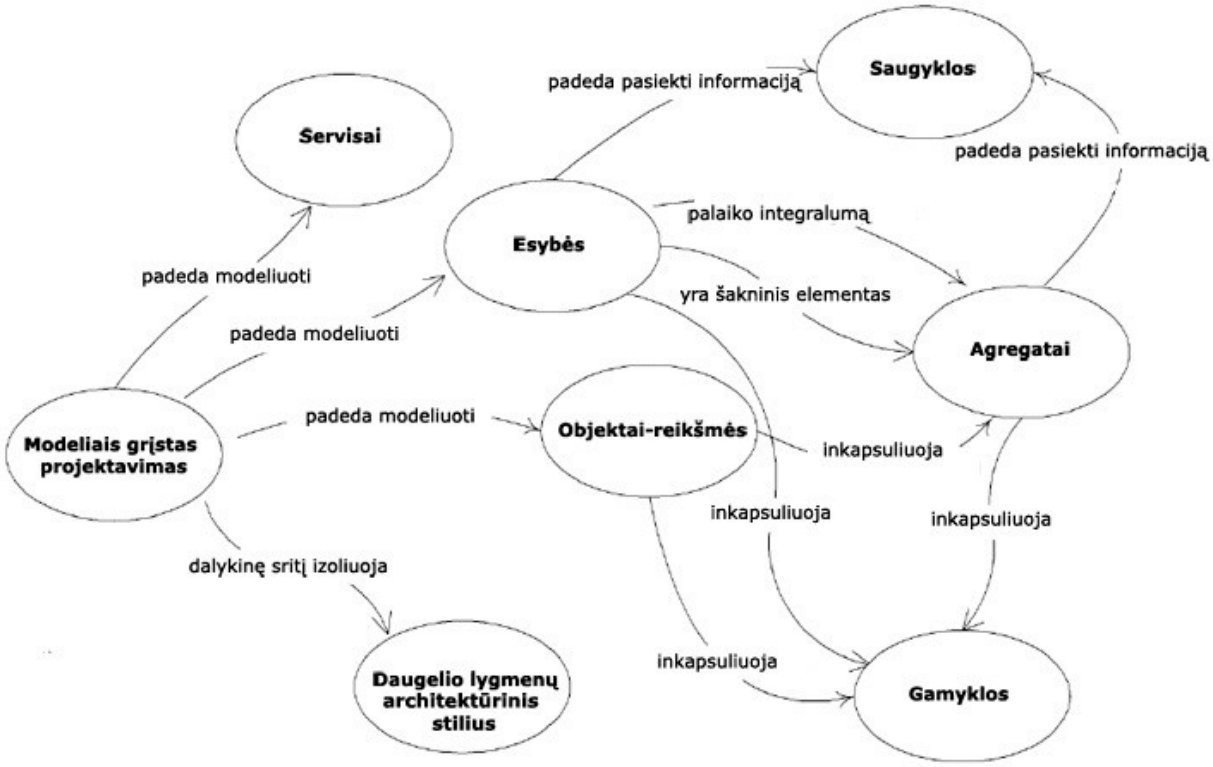
DDD rekomenduojama PS struktūrinio organizavimo schema – daugelio lygmenų architektūrinis stilius (angl. n-tier architecture). Būtent šis stilius geriausiai geba izoliuoti dalykinės srities modelį ir neleisti jam būti transformuojamam. Dalykinės srities lygmuo paliekamas izoliuotas ir nemodifikuotas, tuo tarpu kiti lygmenys yra pridedami realizuojant technologinius sistemos aspektus.

Daugelio lygmenų architektūrinis stilius yra plačiai naudojamas organizacijų informacinėse sistemose. Sistema konstruojama iš tam tikro skaičiaus lygmenų, kurių kiekvienas teikia skirtingas paslaugas. Kiekvienas lygmuo veikia kaip serveris į jį besikreipiančiam lygmeniui, ir kaip klientas, kreipdamasis į kitą lygmenį. Sistemos vartotojas bendrauja tik su pirmu lygmeniu. DDD atveju rekomenduojama sudaryti keturių lygmenų architektūrą:

- Vartotojo interfeiso. Atsakingas už informacijos vartotojui pateikimą ir vartotojo komandų interpretavimą.
- Aplikacijos. Plonas lygmuo, kuris koordinuoja aplikacijos veiksmus. Savyje neturi jokios verslo logikos ar esybių būsenų. Tačiau gali turėti, pavyzdžiui, aplikacijos užduočių progreso būseną.
- Dalykinės srities. Turi visą informaciją apie dalykinė srities modelį. Čia talpinamos esybės, objektai-reikšmės, jų ryšiai, servisai ir kita svarbi dalykinės srities informacija.
- Infrastruktūros. Šis lygmuo atlieka pagalbininko vaidmenį kitiems lygmenims, Bendrauja su duomenų bazėmis, turi daug kitų pagalbinių funkcijų.

2.2.5. Sąvokų tarpusavio ryšiai

Apibendrinant aukščiau pateiktą medžiagą, DDD sąvokas ir jų tarpusavio ryšius galima susieti 2 paveiksle parodytu būdu.



2 pav. DDD sąvokų žemėlapis [Eva03]

2.3. Į dalykinę sritį orientuotas programų sistemų realizavimas

2.3.1. Objektinio programavimo problemos

Dalykinės srities struktūra yra gerai atvaizduojama į objektinės paradigmos klasių egzempliorius – objektus. Tačiau sistemose egzistuoja ne vien statiška struktūra, bet ir pridėtinę vertę kurianti sistemos elgsena. Į dalykinę sritį orientuotame projektavime (DDD) tokia elgsena buvo identifikuota kaip servisas ir patalpinta į dalykinės srities modelį.

Objektiškai orientuota paradigma nesiūlo jokių potogių būdų, kaip būtų galima atvaizduoti sistemos elgseną – sąveiką tarp objektų. Kaip ir dalykinės srities struktūra, taip ir objektų bendradarbiavimas bei sąveika turi struktūrą. Tokios sąveikos yra vartotojo mentalinio modelio ir tuo pačiu dalykinės srities dalis. Objektiškai orientuota paradigma paremtuose sprendimuose nerandamas tokių sąveikų atitikmuo programiniame kode. Pavyzdžiui, tekstų rengyklės savybė patikrinti rašybą. Tai sistemos elgsena sąveikaujanti su redaguojamu tekstu ir žodynu. Tampa

neaišku, kuris objektas turėtų talpinti tokią elgseną: ar redagavimo buferis, ar žodynas, ar globalus rašybos tikrinimo objektas. Dažnai programuotojams tenka išskaidyti algoritmą ir paskirstyti tarp kelių objektų. Dažnai pasirinkdami vieną iš variantų programuotojai sudaro prielaidas neteisingai interpretuoti dalykinės srities elgseną arba didinti sankibą tarp objektų.

2.3.2. Objektinio programavimo evoliucija

Objektiškai orientuoto programavimo paradigma buvo kuriama tam, kad jos pagalba būtų galima naudoti realaus pasaulio objektus programavimo aplinkose, unifikuoti programuotojo ir galutinio vartotojo požiūrių perspektyvas programiniame kode. Objektiškai orientuotai paradigmai pavyko gerai atvaizduoti struktūrą. Tačiau anksčiau minėtos problemos kyla norint rasti atitikmenį realiam pasauliui elgsenos požiūriu. Elgsena yra vienas iš galutinio vartotojo mentalinio modelio atvejų arba dalykinės srities panaudojimo atvejis (angl. use case). Taip pat egzistuoja įvairūs sisteminiai aspektai (tokie kaip saugumas, transakcijos, duomenų saugojimas, žurnalizavimas), kurie sistemos konstravimo prasme panašūs į dalykinės srities elgseną – perkerta daugiau nei vieną sistemoje veikiantį objektą. Dėl šių priežasčių objektinės paradigmos pagrindu buvo sukurta projektavimo ir programavimo technika – aspektiškai orientuotas programavimas (AOP, angl Aspect Oriented Programming) [KLM+97]. AOP įvedė naujų sąvokų, kurių pagalba sistemos elgsena buvo išreikšta lanksčiau. Tačiau tai neleido perteikti dalykinės srities modelio į programinį kodą santykiu 1:1. AOP pagrindu atsirado nauja, į vartotojo mentalinį modelį ir dalykinę sritį orientuota architektūra – „duomenys, kontekstas, sąveika“ (DCI, angl. Data Context Interaction). DCI skirtas tam, kad būtų galima realizuoti galutinio vartotojo arba dalykinės srities modelio roles ir sąveiką tarp jų.

Aspektinis programavimas (angl. Aspect Oriented Programming, toliau AOP) yra projektavimo ir programavimo technika sukurta geresniam programų sistemų turinių atskyrimui (angl. separation of concerns). Programavimo technikos grįstos tik vienu abstrakcijos karkasu (tokios kaip objektiškai orientuotos) dažniausiai sudėtingai sprendžia didelio kompleksiškumo sistemų problemas. Programų sistemoje identifikuojami persikertantys turiniai (dar vadinami aspektais, angl. aspect). Jie sukuriami kiek įmanoma natūralesnėje formoje ir įpinami (angl. woven) kartu su kitais sistemos elementais. Aspektų pavyzdžiai gali būti duomenų saugojimas, sinchronizacija, žurnalizavimas, saugumas ir t.t. AOP suinteresuotas, kad būtų rasti būdai kaip patogiausiu būdu išreikšti aspektus ir supinti juos į vykdomą kodą.

Aspektai perkerta objektinio programavimo modulius. Aspektai supinami kartu į programinį kodą. Pagal turinių atskyrimo moduliariškumo principą [Par72] AOP bando dekomponuoti perkertančius turinius (sisteminį funkcionalumą, angl. crosscutting concerns) į atskirus modulius.

Egzistuoja nemažai skirtingų krypčių po AOP vardu. Dauguma jų yra paremta ant išplėstų objektiškai orientuotų modelių ir daugiausia besikoncentruojančių ties programavimo technikos išreiškimu, plečiamumu, pakartotiniu panaudojimu ir moduliariškumu.

2.3.3. Dalykinės srities funkcionalumo kūrimas Java EE technologinėje platformoje

Java EE platformai yra sukurtas serveryje vykdomas EJB 3.0 komponentų modelis [JSR06]. EJB komponentas inkapsuliuoja programų sistemos dalykinės srities elgseną (angl. business logic). EJB komponentai supaprastina didelių, išskirstytų aplikacijų kūrimą, didina dalykinės srities modelio nemodifikavimo galimybes:

- EJB konteineris pateikia sisteminio funkcionalumo (kaip saugumas, transakcijos ir pan.). Todėl kūrimo metu galima koncentruotis į dalykinės srities modelį. Naują sisteminį funkcionalumą galima pridėti nekeičiant jau parašyto programinio kodo. Java anotacijomis įskiepijamas sisteminis funkcionalumas:
    - Atminties valdymas.
    - Gijų valdymas.
    - Deklaratyvios transakcijos.
    - Deklaratyvus saugumas.
- Komponentai vykdomi serveryje ir yra pilnai atskirti nuo kliento prezentacijos funkcionalumo. Taip atskirti turiniai leidžia geriau izoliuoti dalykinės srities modelį.
- Komponentai yra pernešami (angl. portable).

Egzistuoja dviejų tipų EJB komponentai:

- Sesijos komponentas (angl. session bean). Vykdo kliento užduotį. Gali įgyvendinti žiniatinklio paslaugą (angl. web service).
- Į žinutes orientuotas (angl. message-driven) komponentas. Laukia tam tikro tipo žinutės, pavyzdžiui, iš JMS (angl. Java Message Service) API.

Taip pat egzistuoja duomenų esybės, kaip trečio tipo komponentas, tačiau priskirtas ne EJB, bet JPA [JSR06] (angl. Java Persistence API). Duomenų esybės yra duomenų bazėje automatiškai saugomos dalykinės srities esybės.

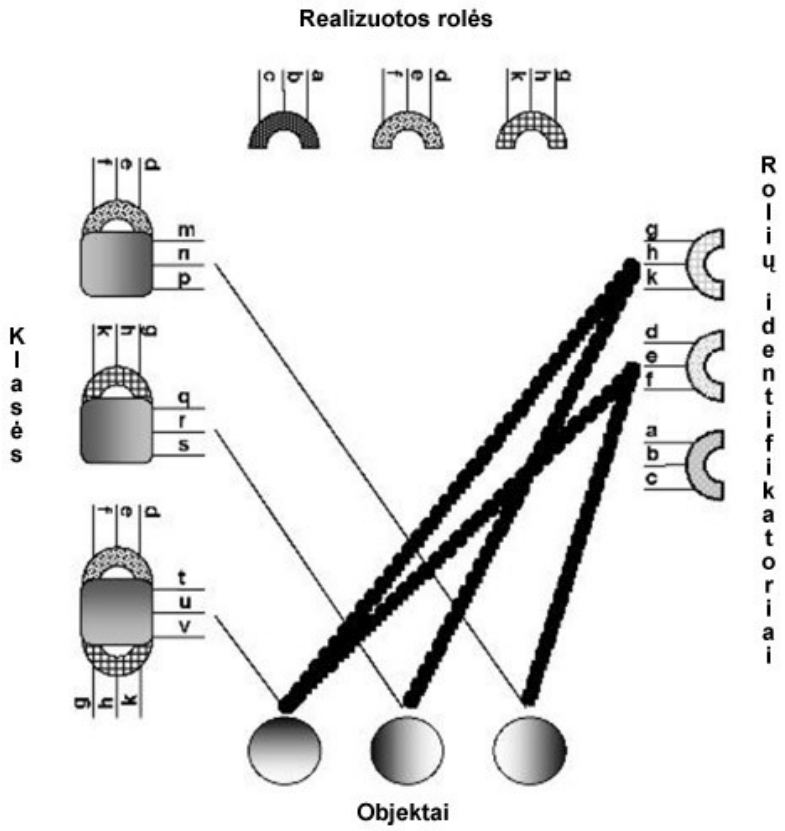
2.4. DCI paradigma

Viena iš svarbių programavimo kalbos užduočių yra sugebėti iliustruoti tai, ką programuotojas turėjo galvoje. Programinio kodo skaitytojas skaitydamas kodą turėtų natūraliai suprasti programuotojo intencijas išreikštas kodu – ką programuotojas ketino šiuo kodu padaryti. Išanalizavus galutinio vartotojo mentalinį modelį, programuotojui būtų galima pateikti įrankį, kuris tą modelį pertektų. DCI architektūra ir yra tas įrankis galutinio vartotojo mentaliniui modeliui, o tuo pačiu ir dalykiniai sričiai perteikti.

Rolių, algoritmų objektų ir sąryšių tarp jų derinys skirtas sustiprinti programinio kodo ir dalykinės srities sąvokų atitikimą yra vadinama DCI (Duomenys, kontekstas, sąveika) architektūra [CB10, RC09, Ree08] (angl. Data Context Interaction).

DCI paradigmos idėja yra atskirti programinį kodą, kuris aprašo sistemos būseną, nuo programinio kodo, kuris aprašo sistemos elgseną. Tam tikslui programinis kodas organizuojamas į tris perspektyvas, kurios koncentruojasi į tam tikrus programinio kodo aspektus:

- Duomenys (angl. Data). Kompiuterinė dalykinės srities reprezentacija. Duomenys talpinami dalykinės srities objektuose, kurie sukuriami iš dalykinės srities klasių.
- Kontekstas (angl. Context). Komunikuojančių objektų, kurie įgyvendina programų sistemos vartotojo komandą, tinklo specifikacija. Kontekstas turi 1:1 sąryšį su panaudojimo atveju (angl. use case). Kontekstas vykdymo aplinkoj reikalingus objektus pašaukia naudojimui panaudojimo atvejo scenarijuje.
- Sąveika (angl. Interaction). Objektų komunikavimo būdo specifikacija, kuomet yra vykdoma vartotojo komanda. Sąveika apibūdina galutinio vartotojo (taip pat ir dalykinės srities) algoritmus rolių terminais. Rolės yra vartotojo mentalinio modelio dalis.



3 pav. Struktūros (klasės) ir algoritmo (rolės) supynimas į objektą [RC09]

Objektiškai orientuota programa yra kompleksinis ir dinamiškas objektų tinklas. Lygiai taip, kaip realaus pasaulio objektai ir sąveika tarp jų. Nagrinėkime teorinį realaus pasaulio objektų ir sąveikos tarp jų pavyzdį. Sakykime, padavėjas dirba restorane. Padavėjas yra kompleksiškas objektas, į kurį galima žiūrėti iš keleto perspektyvų:

- Padavėjas, kuris nupasakoja vakaro meniu ir priima užsakymą.
- Restorano Darbuotojas, turintis savo darbo valandas ir gaunantis atitinkamą atlygį.
- Restorane esantis Asmuo, kur restoranas talpina daugiausiai 200 žmonių.

Jei projektuojant ar konstruojant sistemą klasikiniu objektiškai orientuotu programavimu būtų bandoma parašyti Padavėjo klasę, kuri apimtų visas realiame pasaulyje esančias padavėjo roles, klasė taptų per daug kompleksiška, kad apimtų visas padavėjo perspektyvas. Tačiau objektiškai orientuotos sistemos teoriškai būtent tai ir tūrėtų atlikti.

DCI (angl. Data Context Interaction) atveju visos skirtingos padavėjo pareigos vadinamos rolėmis. Programos vykdymo metu objekto (padavėjo) rolė tampa objekto identifikacija. Panaudojimo atvejo vykdymo metu (pavyzdžiui „Patiekti vyną“) rolė Padavėjas vienareikšmiškai identifikuoja vieną objektą. Galima būtų sakyti, kad gali būti keletas Padavėjų vienam stalui, tačiau jie skirsis savo atsakomybėmis vieno panaudojimo atvejo rėmuose: Pagrindinis Padavėjas, indų

nurinkėjas. Net jei jų atsakomybės būtų identiškos, jie vis tiek būtų kaip individualūs Padavėjo vektoriaus elementai.

DCI panaudojimu atveju Padavėjas yra individualus objektas. Klasikinio objektinio programavimo atveju Padavėjas būtų vienos kasės kompozicijoje kartu su visomis kitomis pareigybėmis (Darbuotojas, Padavėjas, Asmuo). DCI turi tris esminius aspektus:

1. Duomenis. Kiekvienas Padavėjas yra žmogus (arba robotas turintis bazines kalbėjimo ir judėjimo operacijas). Toks elementarus elementas sudaro „kas sistema yra“ dalį.
2. Sąveiką. Elementariesiems sistemos elementams (duomenims) konkrečiu panaudojimo atveju (angl. use case) įsikiepijama tame panaudojimo atvejyje vaidinama rolė. Duomenų ir įskiepytos rolės duetas vadinamas objektu.
3. Kontekstą. Konkretus panaudojimo atvejis (angl. use case), kuriame vyksta sąveikos tarp objektų, objektai orkestruojami. DCI kontekstas yra pirminis projektavimo elementas.

Standartiniame objektiniame projektavime pernelyg supaprastinta taisyklė buvo, kad daiktavardžiai (reikalavimų dokumente) yra objektai, o veiksmažodžiai metodai. Toks skirstymas į dvi dalis natūraliai atitiko du konceptus, kuriuos išreiškia objektinės programavimo kalbos. Objektiškai orientuotos programavimo kalbos išreikšdavo viską per objektus ir metodus tuose objektuose. Taigi, buvo priimta prielaida, kad niekas negali egzistuoti už objekto ribų. Pavyzdžiui, banko sąskaita. Balanso sumažinimas ir lėšų išėmimo operacija sąskaitoje suplakamos į klasės metodus. Abi operacijos yra sistemos elgsena. Tačiau abi yra radikaliai skirtingos. Balanso sumažinimas yra tiktai duomenų charakteristika – kokie duomenys yra. Tuo tarpu lėšų išėmimo operacija vaizduoja duomenų tikslą – ką duomenys daro. Lėšų išėmimas (įtraukiant tranzakcijos semantiką, vartotojo sąveiką, atstatymą, klaidų valdymą, verslo taisykles) stipriai peržengia bet kokio duomenų modelio ribas.

Lėšų išėmimas yra sistemos elgsena keičianti visos sistemos būseną. Tuo tarpu balansu sumažinimas yra tai kas sąskaitą paverčia sąskaita ir susiję tik su konkretaus objekto būsena. Taigi, šios dvi savybės yra labai skirtingos sistemos architektūros, programų sistemos inžinerijos ir priežiūros svarbos prasme. Klasikinis objektinis požiūris suplaka šias savybes į vieną vietą. Jei objektai galvojama, kad turėtų išlikti stabilūs, ir jeigu visas programinis kodas yra juose, tai kur atvaizduoti besikeičiančias dalis? Programos kodo kokybė visada slypėjo jos stabilaus programinio kodo atskirtimi nuo to kas turi potencialą keistis. Tai įgalina atlikti efektyvią sistemos priežiūrą.

DCI architektūra pasiskolino AOP idėjas. DCI taip pat koncentruojasi į išvardintas AOP savybes, tokias kaip turinių atskyrimas. Tačiau esminis skirtumas, kad DCI nesukuria naujų

programavimo technikos sąvokų. DCI, skirtingai nei AOP, koncentruojasi į galutinio vartotojo mentalinio modelio, tuo pačiu ir dalykinės srities sąvokų ir elgsenų modelį programiniame kode. Skirtingai nei aspektai, DCI rolės daugelyje programavimo kalbų randa savo „architektūrinius namus“ naudojantis jau esamomis pragramavimo kalbos priemonėmis. Kitos programavimo kalbos reikalauja tam tikrų patobulinimų. DCI kontekstai sukuria „architektūrinius namus“ programavimo kalboje pačiam panaudojimo atvejui. Tuo tarpu aspektai tik susiporuoja su objektais, kuriems jie taikomi.

2.4.1. Sistemos būsena – duomenys

Sistemos būsena yra išreiškiama duomenų objektų būsena ir ryšiais tarp jų (4 pav. kairioji dalis). Vartotojo mentalinis modelis yra apibrėžtas koncepcinėje schemoje. Schema gali būti realizuota, pavyzdžiui, UML klasių diagramomis be veiklos. Iš duomenų klasių yra kuriamos objektų struktūros, kurios atitinka vartotojo koncepcinę schemą. Aktualių duomenų objektai ir sąryšiai tarp jų yra vykdymo aplinkos reiškinys. Vykdymo aplinka (angl. runtime) 4 paveiksle pavaizduota raudona spalva.

2.4.2. Sistemos elgsena – kontekstas ir sąveika

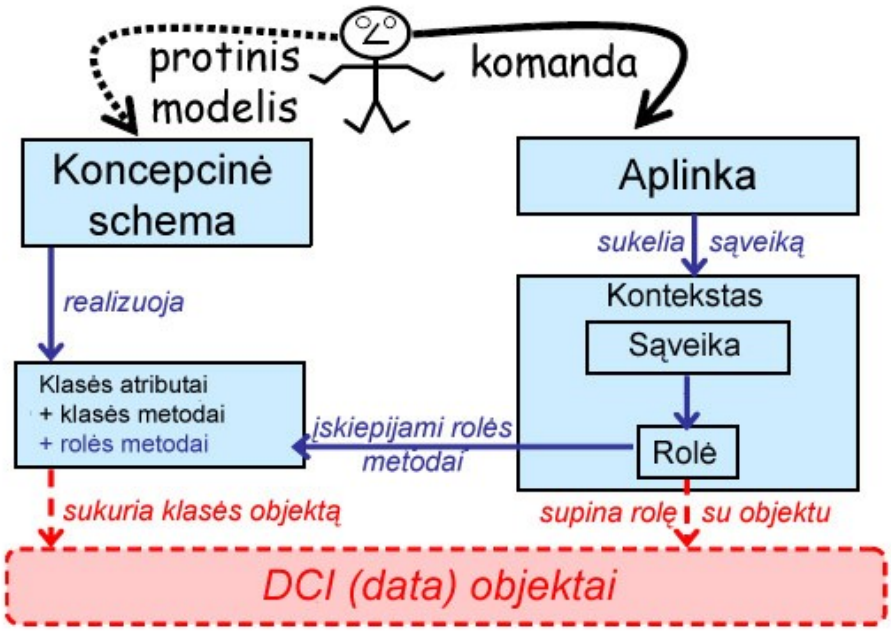
Sistemos vykdymo aplinkos elgsena yra sistemos atsakas į galutinio vartotojo komandas. Vartotojo komanda iškviečia vieno iš sistemos objektų metodą. Tas metodas persiunčia žinutes kitiems susijusiems objektams įvykdydamas vartotojo komandą. Programinis kodas parašytas naudojantis DCI paradigma atskleidžia kodo skaitytojui savo vidinį veikimą. DCI programoje kiekvienas komunikuojančių objektų tinklas yra sukonstruojamas taip, kad atitiktų susijungusių rolių tinklą. Rolė yra savybė leidžianti objektui atlikti papildomas veiklas. Pavyzdžiui, klasės „Asmuo“ objektas galėtų atlikti mokinio, studento, darbuotojo, galiausiai pensininko roles. Vykdymo aplinkos rolių tinklo struktūros sukūrimas ir priežiūra nėra duomenų objektų atsakomybė (skirtingai nei objektiškai orientuotam programavime). Rolių tinklas yra centralizuotas naujame elemente pavadintame kontekstu.

Sistemos elgsena pavaizduota 4 paveikslo dešiniojoje dalyje. Ją sudaro šie elementai:

- Aplinka. Klasė, kuri specifikuoja kaip sistemos operacijos vykdymas parenka atitinkamą kontekstą.
- Kontekstas. Klasė, kuri specifikuoja komunikuojančių objektų tinklą kaip rolių tinklą. Konteksto klasė talpina metodus, kurie supina roles ir objektus vykdymo metu.

- Sąveika. Specifikacija parodanti kaip objektai sąveikauja, kad įgyvendintų sistemos operacija išreikštą rolėmis (rolių tinklu).

Metodai-rolės yra įskiepijami į duomenų klases, ir duomenų klasės jų neperdengia. Programinio kodo skaitytojas gali pasitikėti, kad objektai nedalyvauja jokiuose polimorfiniuose ryšiuose ir neįvyks jokių paslėptos logikos veiksmų. Taigi, programinis kodas turėdamas konteksto, rolių ir duomenų objektų sąvokas išreiškia dalykinės srities statinę struktūrą (būsenos modelį) ir elgseną.



4 pav. DCI architektūra [Ree08]

2.4.3. DCI paradigma vykdymo metu

Apie DCI paradigmą galima galvoti kaip apie įvykiais grįstą (angl. event-driven) paradigmą, kur koks nors įvykis sukelia panaudojimo atvejo vykdymą. Įvykiai yra vadinami trigeriais. Trigeriai yra apdorojami aplinkoje, kurioje yra naudojama DCI paradigma. Tokios aplinkos pavyzdys gali būti kontroleris (angl. controller) įprastoje MVC (Model View Controller) architektūroje arba bet koks sisteminis programinis kodas.

Trigeris iššaukia DCI konteksto (angl. context) objekto inicijavimą. Konteksto objekto tipas pasirenkamas pagal tai, koks panaudojimo atvejis turi būti vykdomas. Pavyzdžiui, bankomatas gali turėti skirtingas konteksto klases tokiems panaudojimo atvejams kaip pinigų pervedimas, išėmimas, įdėjimas, balanso parodymas. Restorano pavyzdžiu skirtingos konteksto klasės gali būti skiriamos atsiskaitymo su klientais, klientų aptarnavimo atvejais. Kai tik aplinka inicijuoja konteksto objekto

sukūrimą, tuoj pat yra vykdomas konteksto objekto atitinkamas metodas pradedantis vykdyti konkretų panaudojimo atvejį.

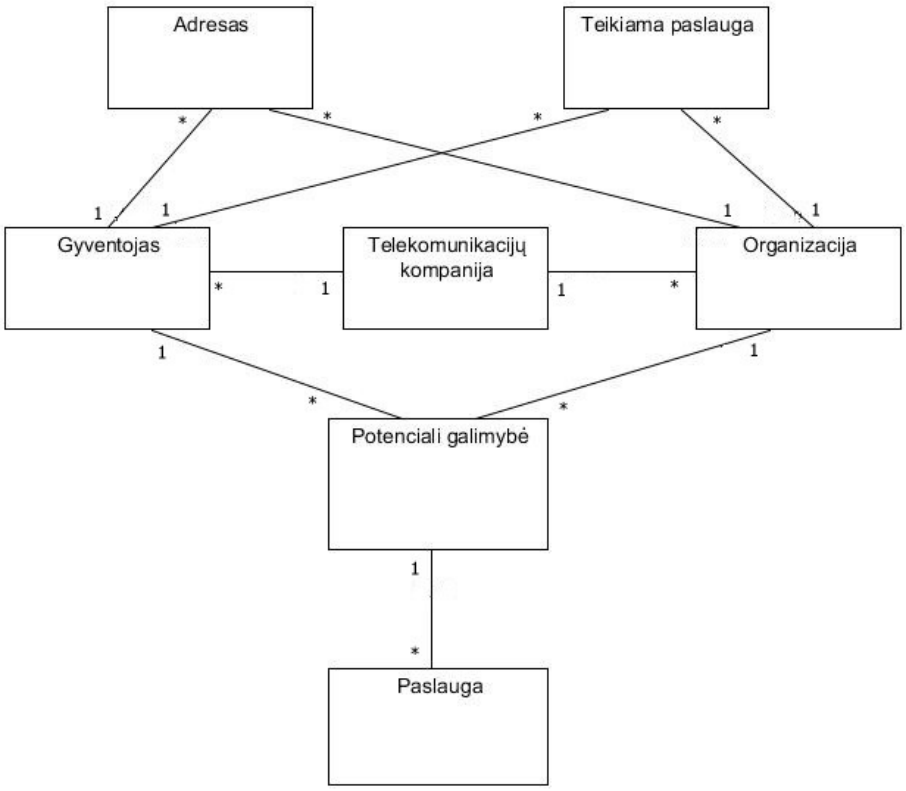
Kiekvienas DCI kontekstas apibrėžia kokios rolės dalyvaus panaudojimo atvejo vykdyme. Taip pat kontekstas yra atsakingas už dalykinės srities objektų ir rolių surišimą. Visas DCI mechanizmo vykdymo modelis pažingsniui galėtų būti išreikštas taip:

1. Kontekstas suranda dalykinės srities objektus, kurie bus reikalingi panaudojimo atvejo įgyvendinimui. Dalykinės srities objektai gali būti pasiekiami bet kokiu būdu: jau egzistuojantys aplinkoje, iš duomenų bazės, sukuriami pačiame konteksto objekte.
2. Dalykinės srities objektui kontekstas priskiria roles reikalingas panaudojimo atvejo įgyvendinimui. Dinamiškose programavimo kalbose, tokiose kaip Ruby ir Python, kontekstas tiesiog įskiepija rolių metodus į objektą. Labiau statinėse programavimo kalbose, tokiose kaip C++, Java, Scala, prieš dalykinės srities objekto ir rolių metodų surišimą turi būti atlikti atitinkami paruošiamieji veiksmai – papildomų programinių konstrukcijų formavimas dėl patogaus rolių metodų įskiepijimo į duomenų objektus.
3. Konktekstas iškviečia pirmojo (ir kiekvieno kito) dalykinės srities objekto vaidinamų rolių metodus reikalingus konkretaus panaudojimo atvejo įgyvendinimui.

3. DCI PARADIGMOS SPRENDŽIAMOS PS REALIZAVIMO PROBLEMOS

3.1. Pavyzdinė dalykinė sritis

Panagrinėkime supaprastintą pavyzdį iš realios telekomunikacijų dalykinės srities. Dalykinės srities modelis (angl. Domain Model) pavaizduotos 5 paveiksle.

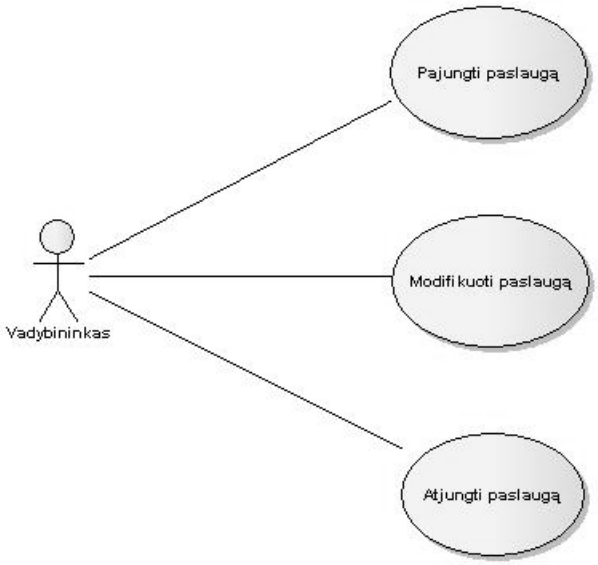


5 pav. Pavyzdinės dalykinės srities modelis

Modelyje egzistuoja dvi pagrindinės dalykinės srities esybės: Gyventojas ir Organizacija. Abi esybės yra skirtingo tipo telekomunikacijų kompanijos klientai. Esybės logiškai atskirtos todėl, kad organizacija turi daug sudėtingesnę, savitą struktūrą. Organizacija gali būti, pavyzdžiui, trijų lygių struktūra – įmonių grupė turinti jai priklausančias įmones, o kiekviena įmonė gali turėti jai priklausančius padalinius. Gyventojas yra atominė esybė. Abiejų tipų klientai turi jiems priklausančius adresus, kuriais yra teikiamos telekomunikacinės paslaugos. Tiek gyventojams, tiek organizacijoms telekomunikacinių paslaugų pardavėjai gali registruoti naujas pardavimines potenciales galimybes. Prie potencialios galimybės yra prisegamos siūlomos

pajungti/modifikuoti/atjungti paslaugos. Telekomunikacinių paslaugų pardavėjams suderinus pardavimo pasiūlymą su klientu siūlomos paslaugos patampa klientui teikiamomis paslaugomis.

Toks dalykinės srities esybių ir ryšių tarp tų esybių pavaizdavimas vadinamas dalykinės srities modeliu. Dalykinės srities modelis apibūdina statinę dalykinės srities struktūrą. Kitaip, tai atitinka „kas sistema yra“ koncepciją (angl. what the system is). Šiame dalykinės srities modelyje trūksta sistemos dinaminės išraiškos – tai ką sistema daro (angl. what the system does). Sistemos elgsena parodyta 6 paveiksle kaip panaudojimo atvejų diagrama.



6 pav. Panaudojimo atvejų diagrama

Supaprastintoje dalykinėje srityje apibrėžiamos trys verslo transakcijos susijusios su telekomunikacinių paslaugų valdymu: paslaugos pajungimas, paslaugos modifikavimas ir paslaugos atjungimas. Šie trys sistemos elgsenos atvejai nėra primityvios vieną dalykinės srities esybę liečiančios operacijos. Kiekvienas iš panaudojimo atvejų visiškam jų įgyvendinimui pasitelkia keletą sistemos objektų keisdamas jų būsenas.

3.2. Projektavimas klasikine objektine paradigma

Klasikinio objektinio projektavimo atveju kiekvienam panaudojimo atvejui būtų bandoma rasti vieta vienoje iš dalykinės srities esybių. Analizuojant dalykinę sritį, jos veiksmažodžius ir daiktavardžius, logines priklausomybes, visi trys pavyzdyje minimi panaudojimo atvejai tikėtina, kad būtų patalpinti kaip metodai į dalykinės srities modelyje identifikuotas esybes.

Keičiantis reikalavimams, augant sistemai, toks transakcijų patalpinimas darytų sistemą stipriai sukibusią (jos vidinių esybių prasme), sunkiai prižiūrimą. Klasės taptų sunkiai skaitomos ir reikalautų didelių kaštų jų atnaujinimui (dėl dydžio ir sankibos su kitomis esybėmis). Dėl stiprios

objektų sankibos pakartotinis kodo panaudojimas tampa netrivialus. Panaudojimo atvejai, tiksliau jų įvykdymą atliekantis programinis kodas būtų išbarstytas po visą dalykinės srities modelį, panaudojimo atvejai „pasimestų“. Toks projektavimo modelis nesutelkia dėmesio į sistemos elgseną. Nors sistemos naudotojo mentaliniame modelyje sistemos elgsena atlieka vieną iš svarbiausių vaidmenų.

Dalykinės srities struktūra (dalykinės srities modelis) yra gerai atvaizduojama į objektinės paradigmos klasių egzempliorius – objektus. Tačiau sistemose egzistuoja ne vien statiška struktūra, bet ir pridėtinę vertę kurianti sistemos elgsena – panaudojimo atvejai. Klasikinis objektinis programų sistemų realizavimas nesiūlo jokių potogių būdų, kaip būtų galima atvaizduoti sistemos elgseną – sąveiką tarp objektų. Kaip ir dalykinės srities struktūra, taip ir objektų bendradarbiavimas bei sąveika turi struktūrą. Tokios sąveikos yra sistemos naudotojo mentalinio modelio ir tuo pačiu dalykinės srities dalis. Klasikiniu objektiniu projektavimu paremtuose sprendimuose nerandamas tokių sąveikų atitikmuo programiniame kode. Pavyzdys galėtų būti tekstų rengyklės savybė patikrinti rašybą. Tai sistemos elgsena sąveikaujanti su redaguojamu tekstu ir žodynu. Tampa neaišku, kuris objektas turėtų talpinti tokią elgseną: ar redagavimo buferis, ar žodynas, ar globalus rašybos tikrinimo objektas. Dažnai projektuotojams/programuotojams tenka išskaidyti algoritmą ir paskirstyti tarp kelių objektų. Dažnai pasirinkdami vieną iš variantų projektuotojai/programuotojai sudaro prielaidas neteisingai interpretuoti dalykinės srities elgseną arba didinti sankibą tarp objektų.

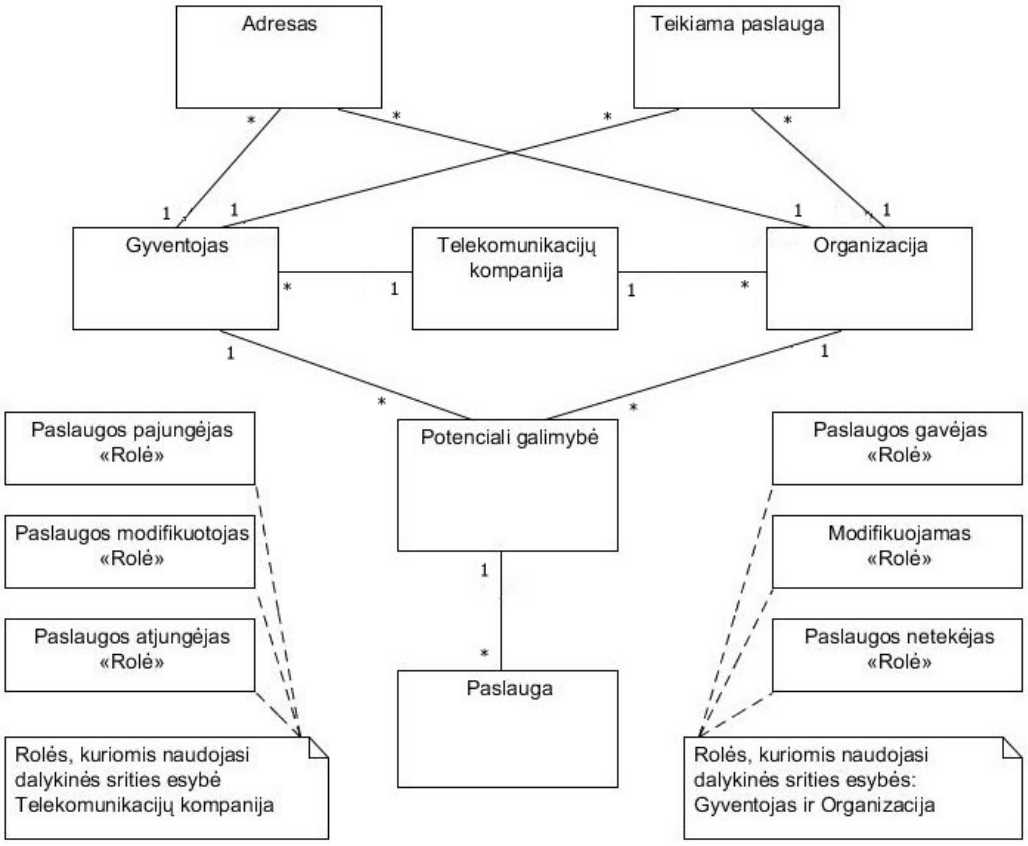
3.3. Projektavimas servisais

Kitas klasikinis pavyzdys, kaip objektinio projektavimo būdu rasti „architektūrinius namus“ sistemos elgsenai, galėtų būti į E. Evans knygoje [Eva03] apibrėžtas dalykinės srities modelio elementas – servisas. Analizuojant dalykinę sritį identifikuojamos dalykinės srities elgsena (servisai). Sąvokos „servisas“ idėja dažnai panaudojama kaip sistemos elgsenos (panaudojimo atvejų) „architektūriniai namai“. Nagrinėjamam pavyzdžiui, pasinaudojus dalykinės srities servisais, kaip projektavimo elementais, būtų sukurti trys statiniai servisai, kuriuos iškviestų servisais pasinaudojantis klientas. Toks projektavimo būdas išsprendžia klasikinio objektinio projektavimo problemas susijusias su kompleksiškomis, didelėmis klasėmis, mažina sistemos projektavimo elementų – klasių sankibą, panaudojimo atvejus realizuojantis programinis kodas nebūtų fragmentuojamas per daugelį klasių – viskas įgyvendinta servisais.

Sistemos evoliucinėje perspektyvoje, augant dalykinės srities modeliui, didėjant panaudojimo atvejų skaičiui, projektavimas servisais priveda prie M. Fowler [Patterns of EAA] aprašyto

architektūrinio projektavimo anti-šablono – anemiško dalykinės srities modelio (angl. Anemic Domain Model). Sistemos elgsenos projektavimas servisais, dalykinės srities esybes paliekant be elgsenos, nutolina nuo esminės objektinio programavimo idėjos, kad objektas savyje turėtų talpinti tiek būseną, tiek elgseną. Pernelyg didelis susikoncentravimas į servisus grąžina žingsniu atgal – į procedūrinio stiliaus projektavimą kartu su visomis šio stiliaus problemomis.

3.4. Projektavimas naudojant DCI paradigmą



7 pav. Dalykinės srities modelis praturtintas rolėmis

Nagrinėjamos dalykinės srities pavyzdį projektuojant DCI paradigma pradedama nuo visų DCI paradigmai reikalingų elementų identifikavimo dalykinėje srityje:

- Duomenų objektų (angl. data).
- Rolių, kurias atlieka duomenų objektai (angl. role).
- Kontekstų (angl. context) arba kitaip – panaudojimo atvejų.
- Sąveikų (angl. interaction) tarp duomenų objektų ir rolių.

3.4.1. Duomenys

Paprastai visos dalykinės srities esybės DCI atveju identifikuojamos kaip duomenų (angl. data) objektai. Taigi, visos dalykinės srities modelio (7 paveikslas) esybės (bet ne rolės) tampa duomenų objektais. Skirtumas nuo klasikinio OO projektavimo, kad klasės neturės metodų, kurie realizuotų dalykinės srities transakcijas (elgseną). Tačiau duomenų objektai gali turėti primityvius dalykinės srities modeliui priklausančius metodus. Tokie metodai turėtų modifikuoti tik paties duomenų objekto atributus, neturint jokių nuorodų į kitus dalykinės srities objektus.

3.4.2. Kontekstai

Anksčiau (6 paveiksle) pateikta panaudojimo atvejų diagrama išskiria 3 panaudojimo atvejus. Projektuojant pagal DCI idėjas, kiekvienas iš panaudojimo atvejų tampa kontekstu. Tokiu būdu atsiranda 3 kontekstai:

- Paslaugos pajungimo kontekstas.
- Paslaugos modifikavimo kontekstas.
- Paslaugos atjungimo kontekstas.

3.4.3. Rolės

Identifikavus DCI kontekstus – panaudojimo atvejus, atliekama kontekstų analizė išsiaiškinant kokie dalykinės srities objektai dalyvauja panaudojimo atvejo įgyvendinime ir kokias roles konkrečiuose kontekstuose jie atlieka (7 paveikslas). Analizuojant nagrinėjamo pavyzdžio 3 su paslaugos atjungimu, pajungimu ir modifikavimu susijusius kontekstus identifikuojama, kad 2 dalykinės srities esybės: gyventojas ir organizacija atlieka skirtingas roles kiekvieno panaudojimo atvejo įgyvendinime:

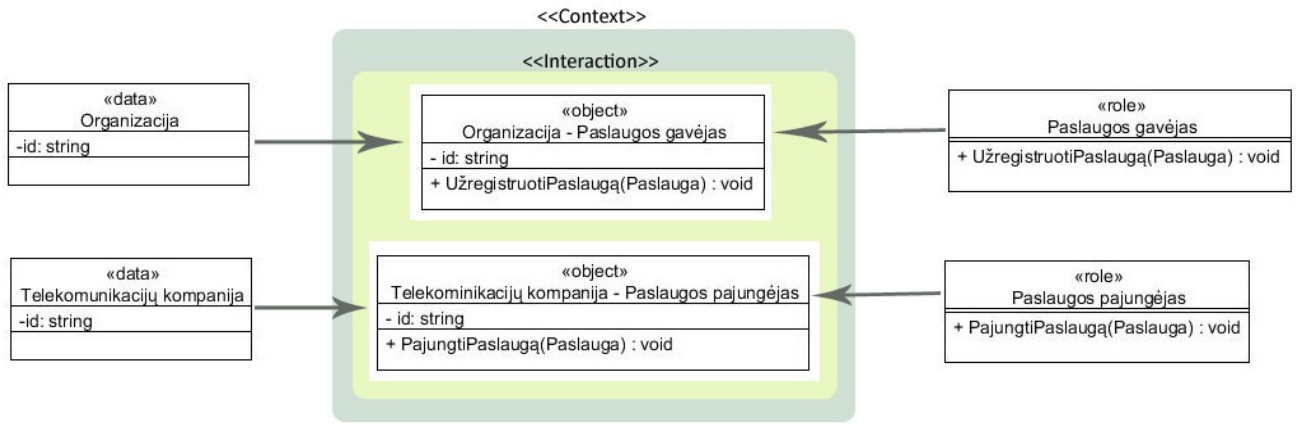
- Paslaugos pajungimo kontekste – paslaugos gavėjo rolė.
- Paslaugos modifikavimo kontekste – modifikuojamojo rolė.
- Paslaugos atjungimo kontekste – paslaugos netekėjo rolė.

Tiek gyventojas, tiek organizacija atitinkamuose kontekstuose gali atlikti gavėjo, modifikuojamojo arba netekėjo rolę. Tuo tarpu telekomunikacines paslaugas teikianti kompanija atitinkamuose panaudojimo atvejuose atlieka:

- Paslaugos pajungėjo rolę.
- Paslaugos modifikuotojo rolę
- Paslaugos atjungėjo rolę.

3.4.4. Sąveika

DCI duomenų objektai, kontekstai ir rolės objektiškai orientuotose programavimo kalbose yra projektuojami klasėmis. Programų sistemos vykdymo metu atsiradus įvykiui apie pradedamą vykdyti panaudojimo atvejį yra inicijuojamas DCI konteksto vykdymas. Pavyzdžiui, vykdant paslaugos pajungimo panaudojimo atvejį klientui organizacijai, būtų inicializuojamas kontekstas „Paslaugos pajungimas“. Konteksto sąveikos (angl. interaction) metode dalykinės srities DCI duomenų objektui „Organizacijai“ būtų dinamiškai įskiepijamas „intelektas“, kitaip elgsena, iš rolės „Paslaugos pajungėjas“. Tokiu būdu gaunamas inkapsuliuotas objektas turintis būseną iš dalykinės srities duomenų objekto ir elgseną iš rolės, kurią duomenų objektas atlieka tam tikrame kontekste. DCI sąveikos proceso pavyzdys pavaizduotas 8 paveiksle.



8 pav. DCI paradigmos esybių, rolių ir konteksto sąveika

3.5. DCI paradigmos sprendžiamos problemos

Keičiantis reikalavimams, augant programų sistemai ilgailaikėje evoliucijoje dalykinės srities struktūros ir elgsenos atskyrimas leidžia programų sistemos programiniam kodui (o taip pat ir kitiems modeliams) išlikti lengvai skaitomam, prižiūrimam ir keičiamam [Obe10].

Pakartotinis programinio kodo panaudojimas naudojant DCI tampa lengvai įgyvendinamas, patogus, trivialus [CB10, Obe10]. Projektuojant ir konstruojant programų sistemą sukuriamos naujos rolės, kontekstai, sąveikos, kurios įgyvendina konkrečius panaudojimo atvejus. Visi šie programinio kodo konstruktai vėliau gali būti pakartotinai panaudojami kitų panaudojimo atvejų įgyvendinime. Pavyzdžiui, 8 paveiksle yra pavaizduota organizacijos ir paslaugos pajungėjo rolės sąveika „Paslaugos pajungimo“ kontekste. Tą pačią rolę galime panaudoti kitame kontekste kitai dalykinės srities duomenų klasei – paslaugos pajungėjo rolę įskiepyti klientui-gyventojui. DCI pakartotiniam kodo panaudojimui pateikia visą reikalingą infrastruktūrą.

DCI pirminis projektavimo elementas nuo kurio prasideda projektavimas yra panaudojimo atvejis. Kiekvienas programų sistemoje esantis panaudojimo atvejis DCI architektūroje yra išreikštinai sukonstruotas kaip kontekstas. Kontekste esanti saveika yra panaudojimo atvejo įgyvendinimo algoritmas. Tokiu būdų panaudojimo atvejį įgyvendinantis programinis kodas yra neišbarstomas po visą dalykinės srities modelį [CB10]. Panaudojimo atvejį įgyvendinantis kodas yra vienoje ir konkrečioje vietoje – kontekste.

DCI sprendžia ir anemiško dalykinės srities modelio (tuo tarpu ir objektų inkapsuliavimo) problemą [Obe10]. DCI objektas dalyvaujantis panaudojimo atvejo įgyvendinime yra inkapsuliuotas būsena, kuri atėjo iš dalykinės srities duomenų esybės, ir elgsena, kuri atėjo iš įskiepytos rolės. Tokiu būdu išvengiama procedūrinio stiliaus naudojant servisus ir grįžtama prie vienos iš esminių objektinio programavimo idėjų – objektų inkapsuliacijos.

DCI architektūra suteikia patogų būdą kaip atvaizduoti sistemos elgseną – sąveiką tarp objektų. Panaudojant DCI architektūrą dalykinės srities elgsena programiniuose konstruktuose randa savo „architektūrinius namus“ – roles ir kontekstus.

4. PSI PROCESŲ MODIFIKAVIMAS DĖL DCI PARADIGMOS PANAUDOJIMO

DCI paradigmos panaudojimas programų sistemų kūrimo procese daro tam tikras kūrimo proceso modifikacijas. DCI paradigma iš esmės remiasi objektinės paradigmos idėjomis, nes programų veikimas grįstas inkapsuliuotais objektais ir projektuojama tais pačiais (kaip ir objektinėje paradigmoje) programavimo kalbų konstruktais.

Daugelyje rinkoj populiariausių programų sistemų kūrimo procesų remiasi panaudojimo atvejų (angl. use case) analize ir visi jie grįsti objektinės analizės ir projektavimo, o taip pat ir objektiškai orientuoto programavimo OOP (angl. Object Oriented Programming) idėjomis.

Praktikoje naudojama populiari Unifikuotų procesų (UP, angl. Unified Process) šeima. UP sudaro tokios realizacijos kaip RUP [Rat98] (angl. Rational Unified Process), OpenUP [Ecl94], Comet [Com07], ICONIX [Ico07] ir keletas kitų kūrimo procesų. Ši programų sistemų kūrimo procesų šeima be išimčių remiasi G. Booch knygoje [Boo93] suformuota programų sistemų inžinerijos požiūriu, kai sistema traktuojama kaip aibė tarpusavyje veikiančių objektų. Toks požiūris vadinamas objektine analize ir projektavimu (angl. Object Oriented Analysis and Design, OOAD).

Objektinės analizės metu taikomos objektinio modeliavimo technikos ir notacijos (pvz. UML, angl. Unified Modeling Language), programų sistemos funkcinių reikalavimų analizei atlikti. Objektinio projektavimo metu yra plėtojami analizės metu sukurti modeliai tam, kad būtų paruoštos programų sistemos realizavimo specifikacijos. Analizės metu dėmesys kreipiamas į sistemos elgseną, tuo tarpu projektavimo metu daugiau dėmesio sutelkiama į sistemos struktūrą.

Detaliau nagrinėjant objektinės analizės procesą, procesą galima suskirstyti į šiuo žingsnius:

1. Panaudojimo atvejų identifikavimas. Analizė pradedama iš sistemos reikalavimų identifikuojant panaudojimo atvejus ir kiekvienam panaudojimo atvejui detalizuojant įvykių seką.
2. Funkcinių testavimo atvejų specifikavimas. Testavimo atvejai analizės metu reikalaingi tam, kad vėliau būtų galima patikrinti, kad realizacija yra pilna ir korektiška.
3. Panaudojimo atvejuose naudojamų klasių identifikavimas ir sąveikos tarp jų ir programų sistemos naudotojų pavaizdavimas.

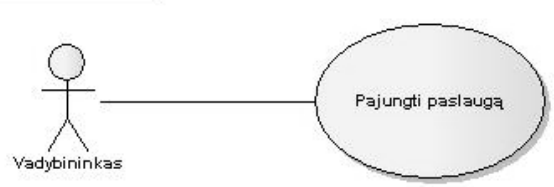
Objektinio projektavimo procesas turi šiuos žingsnius:

1. Analizės metu kiekvienai identifikuotai klasei priskiriamos atsakomybės ir išvardijamos klasės, su kuriomis ta klasė sąveikauja. Tai gali būti atlikta naudojantis CRC (angl. Class Responsibility Collaboration) [BC89] kortelėmis.
2. Statinė projekto struktūra yra pavaizduojama klasių diagrama.
3. Dinaminiai programų sistemos aspektai pagrindiniams panaudojimo atvejams pavaizduojami sąveikų diagramomis.

4.1. Projektavimo procesas projektuojant klasikine objektine paradigma

Anksčiau nagrinėtam telekomunikacinės dalykinės srities pavyzdžiui gali būti pritaikomas OOAD procesas, kuris yra pagrindas visai rinkoje naudojamų, panaudojimo atvejais grįstų, populiarių procesų šeimai. Pavyzdžiui bus pritaikyta 3 objektinės analizės ir 3 objektinio projektavimo žingsniai. Einant per visus proceso žingsnius nagrinėjamas vienas iš 6 paveiksle nurodytų panaudojimo atvejų – paslaugos pajungimas. Tai yra UML panaudojimo atvejų diagrama (angl. use case).

4.1.1. Panaudojimo atvejis



9 pav. Panaudojimo atvejo diagrama

Paslaugos pajungimo panaudojimo atvejo scenarijus prasideda nuo telekomunikacinės kompanijos kliento noro gauti konkrečią paslaugą. Tolimesni scenarijaus žingsniai:

1. Kompanijos pardavimų vadybininkas žinodamas kliento tipą – ar jis gyventojas (fizinis asmuo), ar organizacija, užpildo potencialią galimybę, pasirenka iš siūlomų paslaugų katalogo susijusią paslaugą, pasirenka/įveda kliento adresą, kuriuo bus teikiama ši paslauga.
2. Pardavimų vadybininkas suvedęs/pasirinkęs visą reikalingą informaciją apie potencialią galimybę, paslaugą, klientą inicijuoja paslaugos pajungimo veiksmą.

Po pardavimų vadybininko atliktų scenarijaus žingsnių programų sistema atlieka tolimesnius veiksmus, kad būtų įgyvendintas paslaugos pajungimo panaudojimo atvejis:

1. Programų sistema užfiksavus paslaugos pajungimo iniciavimą:
    a. Surenka reikalingą informaciją iš susijusių objektų.
    b. Atlieka techninių galimybių įdiegti paslaugą tyrimą.

2. Programų sistemai atlikus visus validavimus ir tyrimus, sėkmės atveju:
    a. Atnaujinama informacija susijusi su kliento finansiniu potencialu kliento objekte.
    b. Atnaujinami statusai potencialioj galimybėj.
    c. Suformuojamas dalykinės srities esybės „Teikiama paslauga“ įrašas. Toks įrašas priskiriamas klientui ir reiškia, kad konkrečiam klientui, konkrečiu adresu yra teikiama tokia paslauga.
3. Programų sistemai atlikus visus validavimus ir tyrimus, nesėkmės atveju:
    a. Atnaujinami statusai potencialioj galimybėj.

4.1.2. Funkcinis testavimo atvejis

Pradiniai funkcinio testavimo atvejai sudaromi tam, kad vėliau būtų galima patikrinti programų sistemos elgsenos pilnumą ir korektiškumą. Nagrinėjamo panaudojimo atvejo testavimo atvejis pavaizduotas 2 lentelėje.

2 lentelė. Funkcinio testavimo atvejo detalizacija

Panaudojimo atvejis	Testuojama funkcija	Pradinė sistemos būsena	Įeiga	Laukiama išeiga
Pajungti paslaugą	Paslaugos pajungimas	Sistemoje egzistuoja klientas ir tam klientui konkrečiu adresu nėra jau teikiama paslauga, kurią dabar norime pajungti	Suvesti/Pasirinkti visi (klientas, adresas, potenciali galimybė, paslauga) funkcijos vykdymui reikalingi duomenys	Klientui konkrečiu adresu sukurtas teikiamos paslaugos įrašas

4.1.3. Identifikuotos klasės ir sąveika

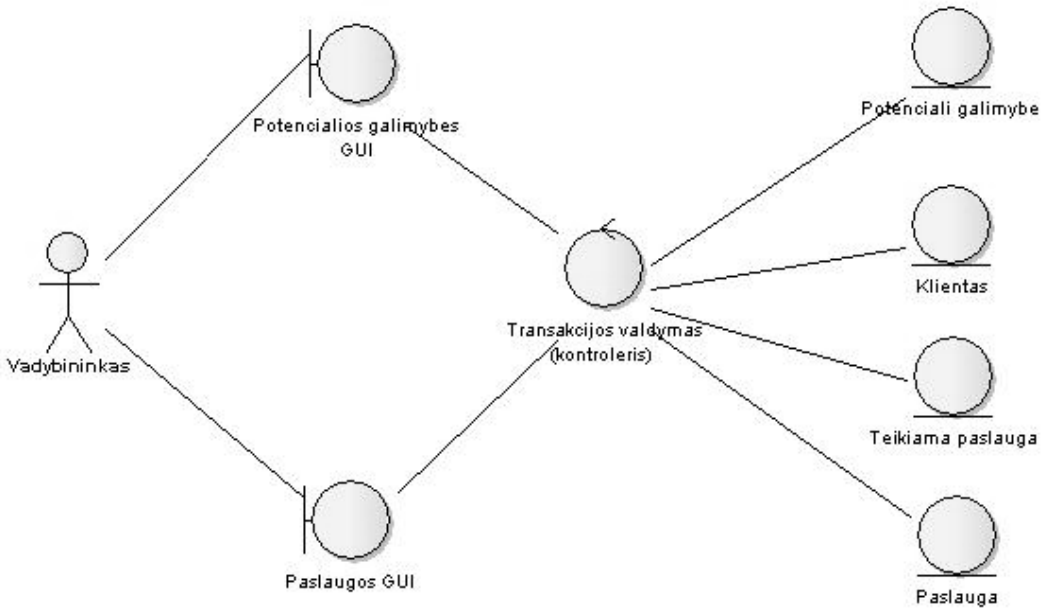
Identifikuotos šios klasės reikalingos panaudojimo atvejo „Pajungti paslaugą“ įgyvendinimui:

- Klasės skirtos sistemos naudotojo sukeliamų įvykių apdorojimui:
    - Potencialios galimybės GUI
    - Paslaugos GUI
- Klasės – dalykinės srities modelio esybės:
    - Potenciali galimybė
    - Paslauga
    - Klientas (Gyventojas arba Organizacija)

- Teikiama paslauga

- Klasė „Transakcijos valdymas“ validuojanti panaudojimo atvejo įeigą, atliekanti papildomus techninių galimybių tyrimus, suformuojanti panaudojimo atvejo išeigą – atnaujinanti statusus, sukurianti įrašus.

10 paveiksle pavaizduota UML komunikacijos diagrama (angl. communication) parodanti klasių ir programų sistemos naudotojo sąveiką vykdant panaudojimo atvejį.



10 pav. Dalykinės srities objektų komunikacijos diagrama

4.1.4. CRC kortelės

Class Responsibility Collaboration (CRC) kortelė yra konceptualus programų sistemų modeliavimo įrankis arba idėja skirta objektiškai orientuotų programų sistemų projektavimui. Idėja pasiūlyta 1989 m. W. Cunningham ir K. Beck. CRC kortelės panaudojamos tuomet, kai priimami pirmieji programų sistemos projektavimo sprendimai identifikuojant klases ir sąveiką tarp klasių. Klasė vaizduoja panašių objektų rinkinį. Klasės atsakomybė yra tai, ką klasė atlieka arba tai, ką klasė žino. Sąveika yra klasės sąryšis su kita klase reikalingas atsakomybėms atlikti. CRC kortelė vaizduojama lentele, kurioje paprastai būna ši informacija:

1. Klasės pavadinimas.
2. Klasės tėvinė ir vaikinės klasės (jei egzistuoja).
3. Klasės atsakomybės.
4. Kitų klasių vardai, su kuriomis klasė sąveikos, kad įgyvendintų savo atsakomybes.

CRC kortelių naudojimas mažina sistemos projekto kompleksiškumą. Naudojant korteles visas dėmesys sutelkiamas į esminius principus susijusius su klasėmis atsiribojant nuo pernelyg didelio detalumo ir vidinės realizacijos. Taip pat kortelės verčia projektuotoją susilaikyti nuo situacijos, kai klasei priskiriama per daug atsakomybių.

Egzistuoja paplitęs metodas, kad sukuriama tiek CRC kortelių, kiek skaitant reikalavimo specifikaciją randama daiktavardžių. Tokiu būdu veiksmažodžiai tampa klasės atsakomybėmis. CRC kortelių sudarymo seka yra:

1. Identifikuoti klases.
2. Surasti klasių atsakomybes.
3. Apibrėžti klasių sąveikas.

CRC kortelės skirtos priskirti atsakomybes visoms klasėms, kurios dalyvauja panaudojimo atvejo „Pajungti paslaugą“ įgyvendinime. Taip pat nustatoma su kokiomis kitomis klasėmis klasė sąveikauja. Informacija pateikiama 3 lentelėje.

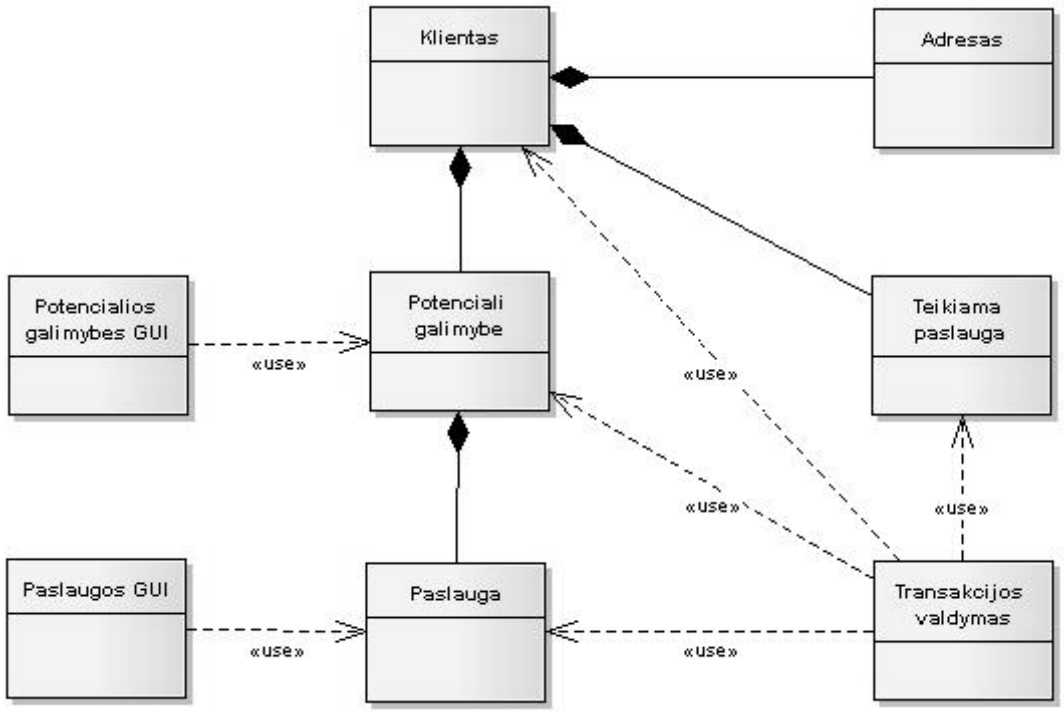
3 lentelė. CRC kortelės

Klasės pavadinimas	Atsakomybė	Sąveika su kitom klasėm
Potencialios galimybės GUI	Leidžia sistemos naudotojui išsaugoti/atnaujinti suvestus arba pasirinktus duomenis apie potencialią galimybę, klientą, adresą.	Potenciali galimybė
Paslaugos GUI	Leidžia sistemos naudotojui išsaugoti/atnaujinti suvestus arba pasirinktus duomenis apie užsakomą paslaugą.	Paslauga
Potenciali galimybė	Saugo informaciją, susijusią su klientu, adresu, apie naują pardaviminę veiklą.	
Paslauga	Saugo informaciją, susijusią su kliento naujai užsakoma paslauga.	
Klientas	Saugo informacija apie kliento kontaktinius duomenis, statusą, finansinį potencialą.	
Teikiama paslauga	Saugo informacija apie kliento jau užsakytas ir naudojamas paslaugas, jų statusus, adresus.	
Transakcijos valdymas	Validuoja potencialios galimybės, paslaugos, kliento duomenis. Atlieka papildomus techninių galimybių tyrimus pasinaudoda	Potenciali galimybė Paslauga Klientas Teikiama paslauga

	trečių šalių teikiamomis tyrimų paslaugomis. Atnaujina potencialios galimybės, paslaugos statusus. Sukuria klasės „Teikiama paslauga“ egzempliorius.	

4.1.5. Statinis sistemos modelis

Statiniui sistemos modeliui pavaizduoti naudojama UML klasių (angl. class) diagrama. 11 paveiksle pavaizduota supaprastinto telekomunikacinės sistemos panaudojimo atvejo „Pajungti paslaugą“ statinis modelis. Pagrindinė klasių struktūra gaunama iš analizės metu identifikuotų klasių ir CRC kortelių stulpelio „Sąveika su kitomis klasėmis“. Klasių diegramoje klasės neturi nei atributų, nei metodų todėl, kad atliekamas architektūrinis projektavimas. Detaliojo projektavimo etape klasėms priskiriami atributai ir metodai.

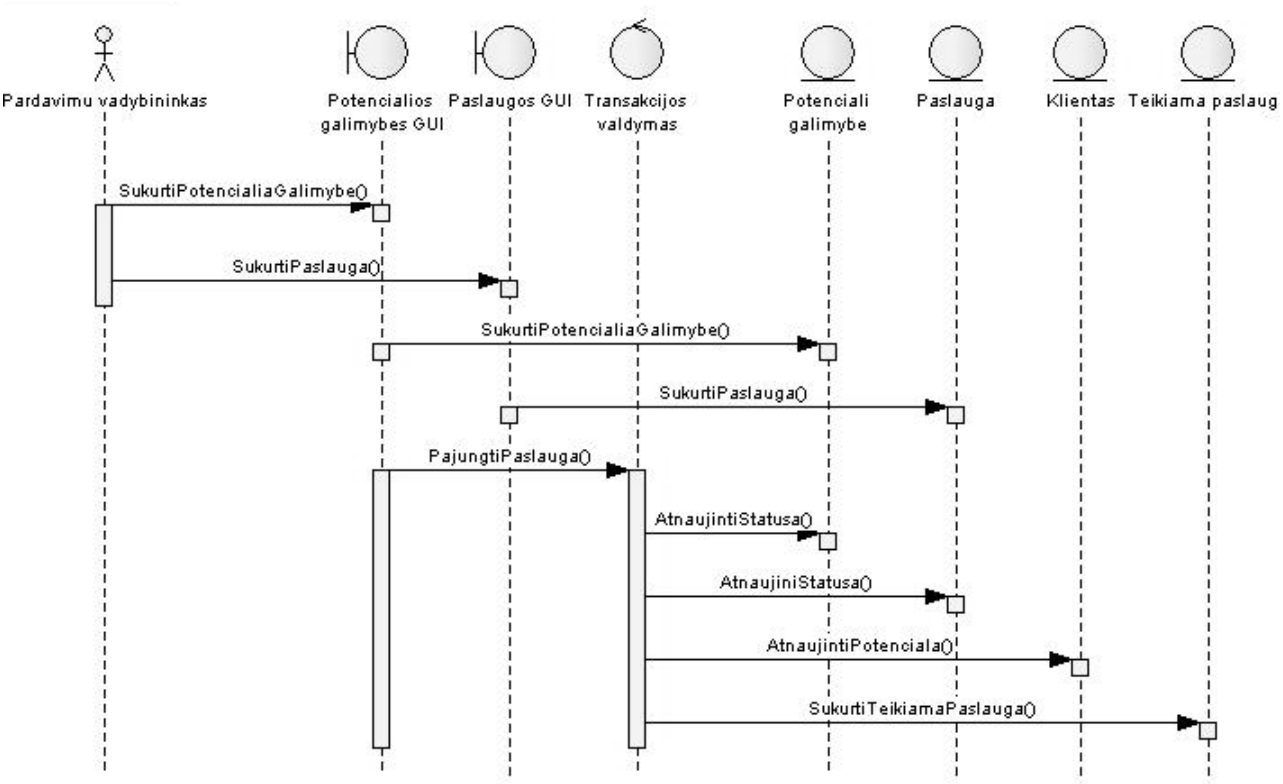


11 pav. Statinis sistemos modelis – UML klasių diagrama

4.1.6. Dinaminis sistemos modelis

Objektinės analizės metu aprašytas panaudojimo atvejis „Pajungti paslaugą“ dinaminiame sistemos modelyje yra išreiškiamas kaip įvairius objektus įtraukianti operacijų seka laiko skalėje

įgyvendinanti panaudojimo atvejį. Dinaminis modelis pavaizduojamas UML sekų (angl. sequence) diagrama (12 paveikslas).



12 pav. Dinaminis sistemos modelis – UML sekų diagrama

4.2. Projektavimo procesas panaudojant DCI paradigmą

DCI paradigma iš esmės yra modifikuota objektinė paradigma, kad geriau įgyvendintų ir inkapsuliuotų dalykinės srities elgseną. DCI paradigmą, gali realizuoti bet kuri objektinio programavimo kalba. Tokiu būdu DCI paradigma gali būti pilnavertiškai naudojama bet kurio anksčiau minėto objektine analize ir projektavimu grįsto ir rinkoje plačiai naudojamo programų sistemų kūrimo proceso rėmuose. Svarbu paminėti, kad tokie programų sistemų kūrimo procesai labai dera su DCI paradigma todėl, kad jų reikalavimų analizės etapas pradedamas nuo panaudojimo atvejų identifikavimo ir analizės. DCI paradigma pradinis projektavimo elementas taip pat yra panaudojimo atvejis – programų sistemos elgsena. Dar daugiau – DCI išreikštinai pateikia architektūrinę vietą programų sistemos elgsenai – tai yra išskirtus kontekstus ir roles.

Taigi, kuo skiriasi OOAD procesas naudojant klasikinį OO ir DCI? Bus nagrinėjamas tos pačios dalykinės srities pavyzdys ir panaudojimo atvejis pereinant visus 6 OOAD žingsnius.

4.2.1. Panaudojimo atvejis

Objektinės analizės procese naudojant DCI paradigmą panaudojimo atvejų identifikavimas (aktorių, jų atliekamų veiksmų), detalių panaudojimo atvejį įgyvendinančių žingsnių aprašymas iš esmės niekuo nesiskiria nuo klasikinio objektinio programavimo naudojimo. Todėl ir telekomunikacinės dalykinės srities nagrinėtas pavyzdys panaudojimo atvejų analizės žingsnyje yra identiškas. Galima teigti, kad ta pati panaudojimo atvejų diagrama (6 paveikslas) ir tie patys panaudojimo atvejo scenarijaus žingsniai tinka ir DCI paradigmos naudojimo atveju.

4.2.2. Funkcinis testavimo atvejis

Kaip ir panaudojimo atvejai, pradinių funkcinių testavimo atvejų analizės žingsnio DCI panaudojimas niekaip nemodifikuoja. Pradinių funkcinių testavimo atvejų identifikavimas turi aukšto lygio tikslą – patikrinti programų sistemos korektiškumą jos vėlesnėse projektavimo ir įgyvendinimo stadijose. Todėl, tokie aukšto lygio tikslai praktiškai negali priklausyti nuo vienokios ar kitokios projektavimo ir realizavimo paradigmos panaudojimo. DCI paradigmos panaudojimo atveju pavyzdinei dalykinei sričiai būtų sukurta funkcinius testavimo atvejus aprašanti 2 lentelė.

4.2.3. Identifikuotos klasės ir sąveika

Vykdant objektinės analizės proceso žingsnį, kuriame konceptualiai identifikuojamos klasės ir sąveika tarp jų, vaizduojama UML komunikacijos diagrama. DCI paradigma atskiria sistemos elgseną nuo struktūros – roles nuo dalykinės srities objektų. Ši paradigmos savybė yra viena iš pagrindinių. Įgyvendinant konkretų panaudojimo atvejį komunikacijos diagramoje klasikinio objektinio projektavimo atveju yra vaizduojamos klasės ir jų sąveika. DCI paradigmos panaudojimo atveju komunikacijos diagramos klasės yra ne klasikinio OOP klasės, o DCI paradigmos objektai – dalykinės srities esybės su inkapsiliuotomis rolėmis. Tokiu būdu komunikacijos diagrama parodo panaudojimo atvejo įgyvendinimą su inkapsuliuotais DCI objektais.

Šį proceso žingsnį dar labiau adaptuojant prie DCI paradigmos naudojimo, galima modifikuoti panaudojant UML notacijos specifikacijoje [Omg07] aprašytą bendradarbiavimo diagramą (angl. collaboration diagram). Bendradarbiavimo diagramomis būtų parodoma kokias roles gali atlikti dalykinės srities esybės konkrečiuose panaudojimo atvejuose (13 paveikslas).

UML bendradarbiavimo diagramoje punktyrinė elipsė atitinka DCI paradigmos konteksto (panaudojimo atvejo) sąvoką. Stačiakampiais žymimos dalykinės srities esybės. Kontekstą ir esybę jungiantis ryšys yra esybės ir rolės surišimas. Ryšio pavadinimas – DCI rolės pavadinimas.

13 pav. UML bendradarbiavimo diagrama rolių modeliavimui

4.2.4. CRC kortelės

Naudojant klasikinį objektinį programavimą objektinės analizės procese klasių atsakomybės buvo išskirstytos klasėms naudojant CRC (angl. Class, Responsibilities, Collaborations) korteles. Tačiau tokių kortelių sudarinėjimas naudojant DCI paradigmą nėra tikslingas. Galima teigti, kad tokios kortelės iš proceso yra eliminuojamos, nes iš esmės pačios klasės neatskleidžia atsakomybių, kurias jos turi. Atsakomybes atskleidžia klasės atliekamos rolės konkrečiuose panaudojimo atvejuose.

DCI naudojimo atveju *Class-Responsibilities-Collaborations* kortelės gali būti pakeistos į panašios koncepcijos idėją aprašytą A. McKean knygoje [MW02] apie objektinį projektavimą. Tai yra *Candidate role-Responsibilities-Collaborations* kortelės. Ant į klases orientuotų kortelių registruojamas klasės pavadinimas, klasės atsakomybės ir kitų klasių pavadinimai, su kuriomis klasė bendradarbiaus, kad įvykdytų savo atsakomybes. Tuo tarpu rolėmis grįstų kortelių koncepcija keičiasi taip: rolės pavadinimas, rolės ir objektų, kurie vaidina tą rolę, atsakomybės, ir rolės, su kuriomis rolė bendradarbiaus, kad įvykdytų savo atsakomybes.

4 lentelė. *Candidate role-Responsibilities-Collaborations* kortelės

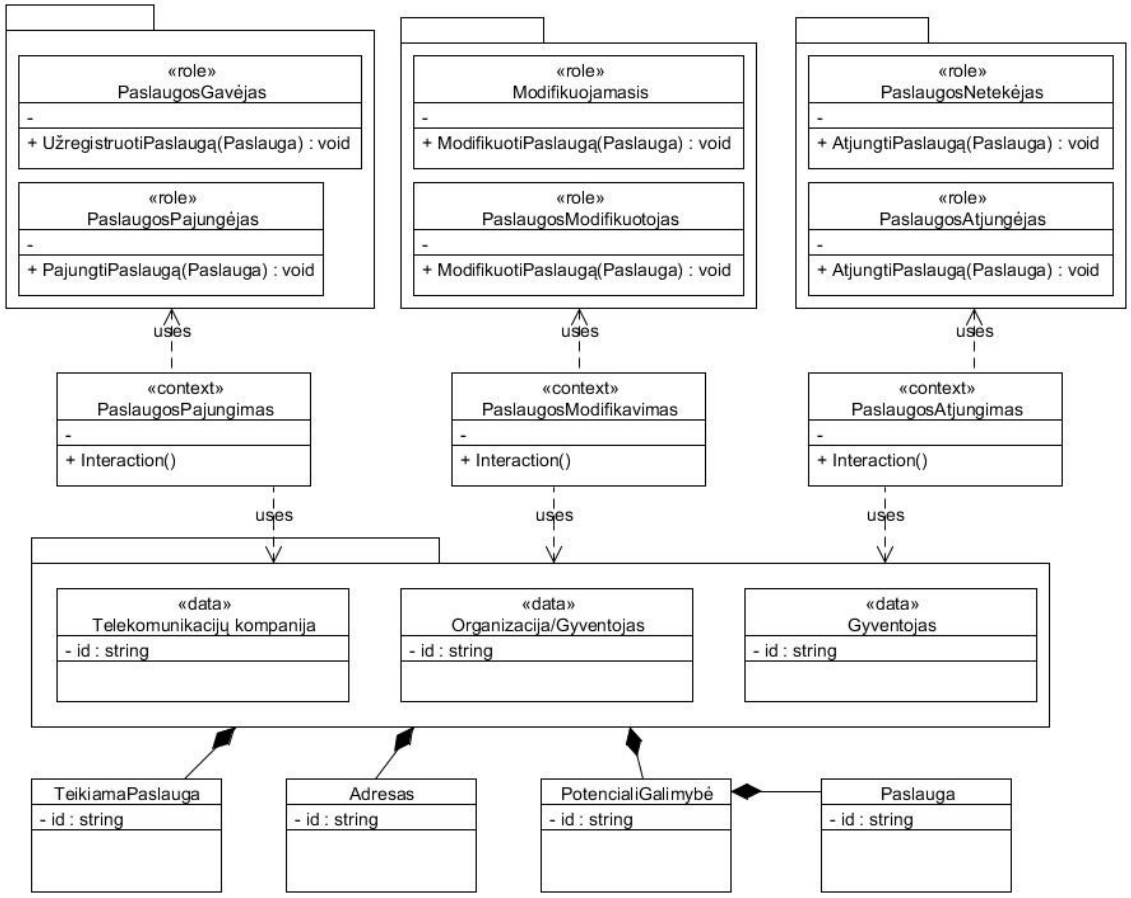
Rolės pavadinimas	Atsakomybė	Sąveika su kitom rolėm/klasėm
Paslaugos pajungėjas	Rolė leidžianti klientui pajungti užsakomą telekomunikacinę paslaugą.	Klientas Teikiama paslauga
Paslaugos atjungėjas	Rolė leidžianti atjungti klientui teikiamą telekomunikacinę paslaugą.	Klientas Teikiama paslauga
Paslaugos modifikuotojas	Rolė leidžianti pakeisti įvairius klientui teikiamos telekomunikacinės paslaugos parametrus pagal kliento pageidavimą.	Klientas Teikiama paslauga
Potenciali galimybė	Saugo informaciją, susijusią su klientu, adresu, apie naują pardaviminę veiklą.	

Paslauga	Saugo informaciją, susijusią su kliento naujai užsakoma paslauga.	
Klientas	Saugo informacija apie kliento kontaktinius duomenis, statusą, finansinį potencialą.	Paslaugos pajungėjas Paslaugos atjungėjas Paslaugos modifikuotojas
Teikiama paslauga	Saugo informacija apie kliento jau užsakytas ir naudojamas paslaugas, jų statusus, adresus.	

4.2.5. Statinis sistemos modelis

Statiniam sistemos modeliui išreikšti panaudota UML klasių diagrama. Tokiu pačiu principu galima modeliuoti ir DCI paradigmos klases į pagalbą pasitelkiant UML klasių diagramų elementą – stereotipą. Kaip teigia šaltinis [BGB05]: „stereotipas yra UML modelio elemento žyma, kuri niekaip kitaip nekeičia modelio elemento, tik nurodo pridėtines modelio elemento charakteristikas, kurios yra bendros modeliuojamos aplikacijos rėmuose“. Taigi, panaudojant UML klasių diagramų stereotipų savybę galima modeliuoti statinį sistemos modelį paremtą DCI paradigma. Tokiu būdu būtų sukuriami 3 nauji stereotipai:

- Dalykinės srities esybių klases žymintis stereotipas („`<<data>>`“).
- Rolių klases, kurios bus įskiepytos į dalykinės srities esybes, žymintis stereotipas („`<<role>>`“).
- Kontekstų – panaudojimo atvejų klases žymintis stereotipas („`<<context>>`“)

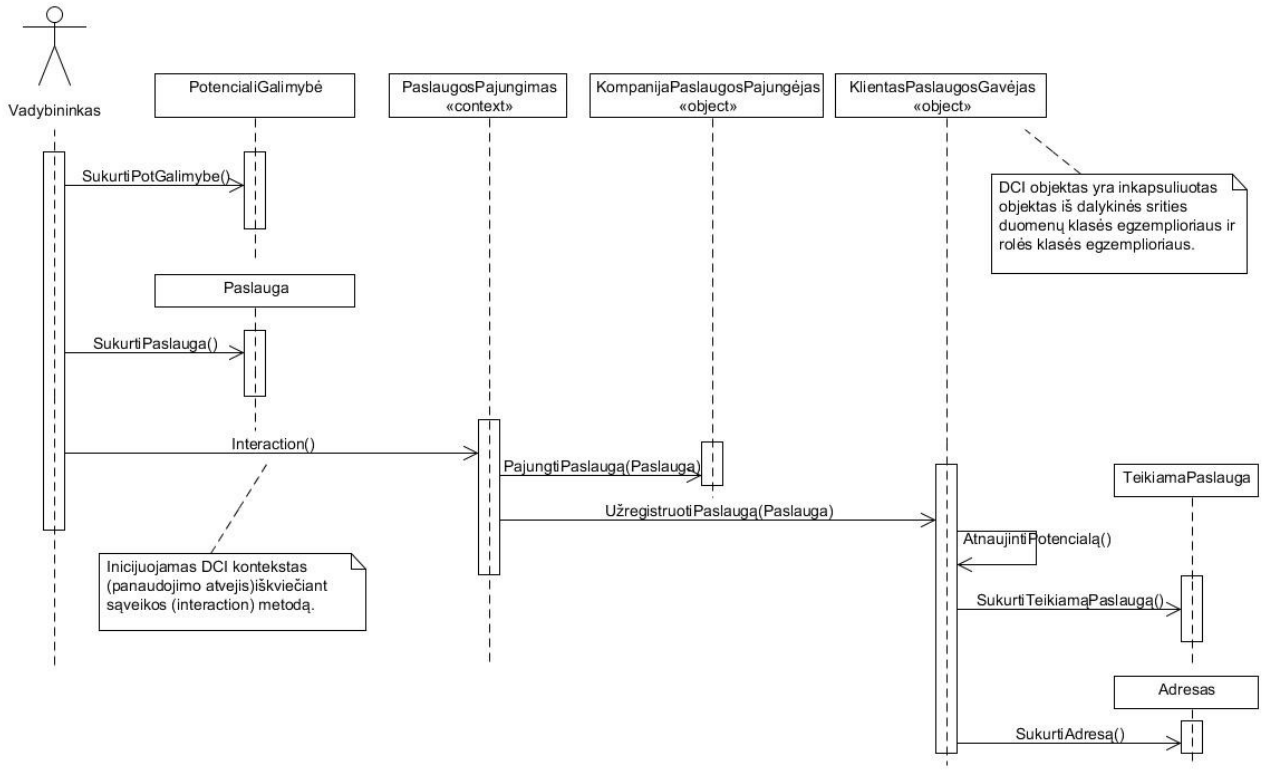


14 pav. Statinio sistemos modelio klasių diagrama

4.2.6. Dinaminis sistemos modelis

Dinaminiam sistemos modeliui pavaizduoti panaudota UML sekų diagrama. Tokiomis pačiomis priemonėmis galima modeliuoti programų sistemos elgseną ir naudojant DCI paradigmą. Verta paminėti, kad sekų diagramos objektas (stulpelis, ne aktorius) DCI atveju nebus vienos klasės egzempliorius programų sistemos vykdymo metu. DCI objektas yra inkapsuliuotas objektas iš dalykinės srities duomenų klasės egzemplioriaus ir rolės klasės egzemplioriaus.

DCI naudojimo atveju iš sistemos naudotojo inicijuoto kontrolerio panaudojimo atvejo vykdymas bus pradėtas kviečiant konteksto objektą. Konteksto objektas savyje turi sąveiką tarp dalykinės srities objektų ir rolių, panaudojimo atvejį įgyvendinantį algoritmą.

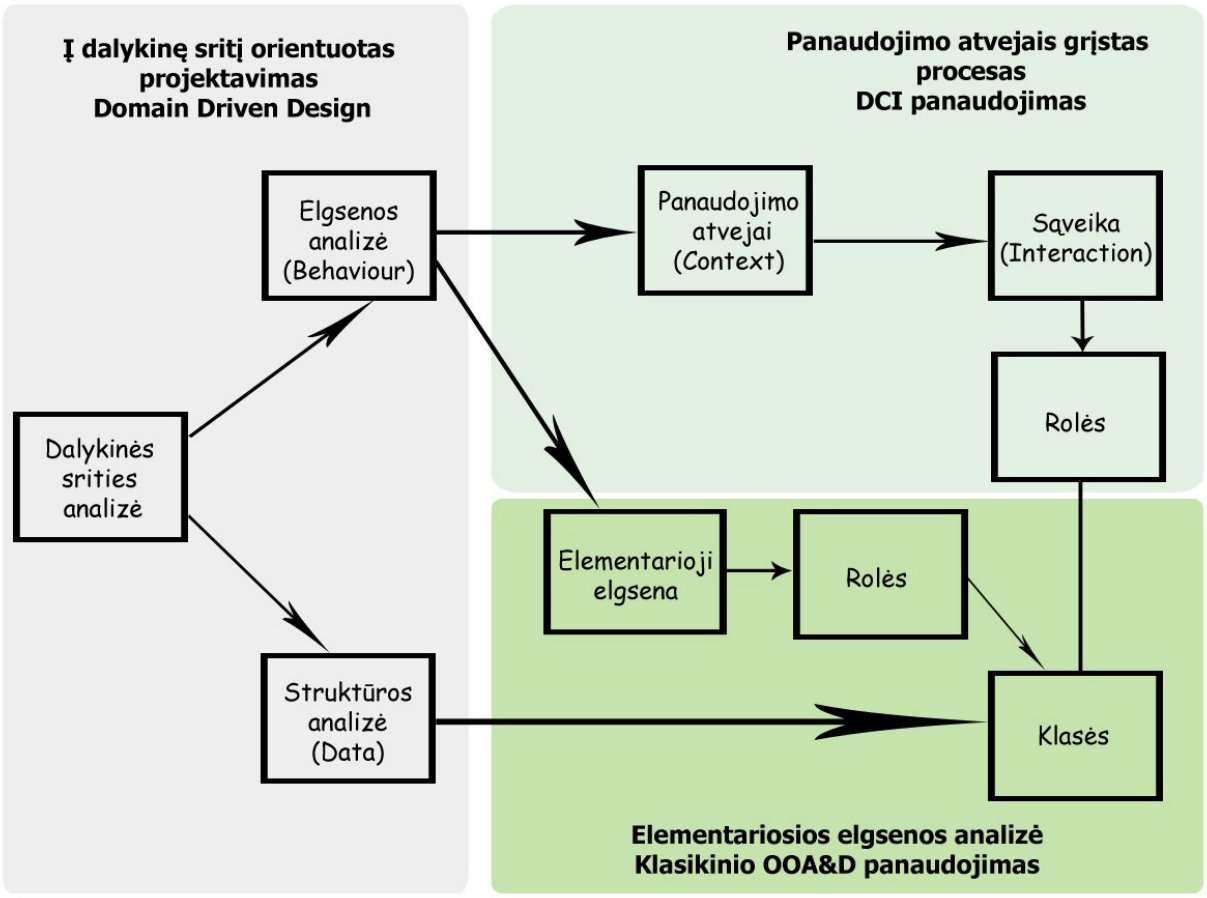


15 pav. Dinaminio sistemos modelio sekų diagrama

4.3. Rolėmis praplėstas programų sistemų kūrimo procesas

Daugumos programų sistemų kūrimas pradedamas (16 pav.) nuo dviejų pagrindinių analizės gijų: dalykinės srities (kaip statinės struktūros, angl. domain analysis) ir elgsenos analizės (angl. behaviour analysis).

Dalykinės srities analizė gali remtis E. Evanso suformuotomis DDD idėjomis [Eva03, AM06]. Čia yra pasineriama į dalykinės srities žodyną diskutuojant su jos ekspertais, identifikuojami pagrindiniai DDD komponentai (esybės, objektai-reikšmės, agregatai, gamyklos, saugyklos, servisai), iš jų sudaromas dalykinės srities modelis. Dalykinės srities modelis formuojamas naudojant įvairius modelio integralumui evoliucijoje palaikyti projektavimo šablonus. Taip sudaryto dalykinės srities modelio esybės ir objektai-reikšmės paverčiami objektinio programavimo konstruktais – klasėmis. Klasės yra statiniai sistemos elementai. Taigi, analizuojant dalykinės srities statinę struktūrą, vykdomas į klases orientuotas projektavimas.



16 pav. Rolėmis praplėsto kūrimo procesai komponentai

Elgsenos analizė išskiriama į dvi atskiras šakas: panaudojimo atvejus (angl. use case) ir elementariąją elgseną.

Elementarioji elgsena išskiriama tam, kad atskirti trivialius elgsenos atvejus, kurie yra tiesiogiai projektuojami klasėmis ir role (t.y. klasės metodu). Elementariosios elgsenos atveju sąvoka rolė naudojama ta prasme, kad objektas vykdymo metu atliks kažkokią rolę – bus naudojamas jo dalykinis metodas. Tačiau rolė yra „lokali“ – ji keičia tik to paties objekto būseną. Pvz, galutinio vartotojo komandų atpažinimas ir jų apdorojimas – atominė (neskaidoma) operacija “pakeisti grafinio objekto spalvą”. Tokia elementari elgsena nesinaudoja daugiau nei viena dalykinės srities esybe ir gali būti logiškai patalpinta (pagal savo prasmę dalykinėje srityje) dalykinės srities esybėje kaip jos metodas. Tokiai elgsenai projektuoti užtenka klasikinių objektinio programavimo sąvokų. DCI paradigmos panaudojimas šiuo atveju būtų perteklinis.

Panaudojimo atvejis yra tam tikra užduočių seka/sąveika (objektų scenarijų kolekcija) skirta pasiekti iš anksto užsibrėžtą tikslą tam tikrame kontekste. Elgsenos analizės metu tokios sekos suteikiančios dalykinės srities veiklai pridėtinę vertę turi būti identifikuotos kaip panaudojimo

atvejai. Pvz, tai galėtų būti pinigų pervedimas iš vienos banko sąskaitos į kitą. Analizuojant panaudojimo atvejį yra identifikuojamos dalykinės srities esybės, kurios dalyvauja panaudojimo atvejo įgyvendinimo scenarijuje. Taip pat išskiriamos rolės, kurias dalykinės srities esybės atlieka konkrečiame panaudojimo atvejyje. Programų sistemos panaudojimo atvejo vykdymo metu, kuris inicijuojamas iškviečiant DCI kontekste esantį sąveikos (angl. Interaction) metodą, į reikalingą dalykinės srities objektą yra įskiepijama objekto atsakomybė tame panaudojimo atvejyje – rolė.

REZULTATAI IR IŠVADOS

Rezultatai

Baigiamajame magistro darbe pasiekti šie darbo rezultatai:

- Pademonstruota kaip panaudojant naują projektavimo, programavimo paradigmą sprendžiamos esminės klasikinio objektinio programavimo problemos.
- Pasiūlytos programų sistemų kūrimo proceso modifikavimo rekomendacijos dėl rolinio modeliavimo panaudojimo visuose sistemos kūrimo etapuose.

Išvados

Išanalizavus įvairius programų sistemų kūrimo procesus ir metodus paaiškėjo, kad rolinio modeliavimo naudojimas sistemos analizės ir projektavimo etapuose programų sistemų kūrimo industrijoje neprigijo dėl konstravimo etapui trūkstamų programinių priemonių programuoti rolėmis. Prieita prie išvados, kad programavimo paradigmos galinčios modeliuoti rolėmis atsiradimas ir panaudojimas sistemos konstravimo etape, rolinį modeliavimą padaro prasmingą visuose programų sistemos kūrimo etapuose. Panaudojant dalykinės srities modeliais grįsto projektavimo idėjas ir minėtą objektionio programavimo paradigmą galima teigti, kad iš esmės yra įmanoma išlaikyti dalykinės srities modelį netransformuotą visame programų sistemos kūrimo procese.

Atlikus objektinio programavimo evoliucijos tyrimą išsiaiškinta, kad šiandien plačiai naudojamas klasikinis objektinis programavimas neturi priemonių kaip inkapsuliuotai išreikšti programų sistemos elgseną – panaudojimo atvejus. Tokia situacija programų sistemos evoliucijos perspektyvoje priveda prie problemų susijusių su sistemos prižiūrimumu, patikimumu, pakartotiniu programinio kodo panaudojimu. Pastarosios problemos gali būti sprendžiamos panaudojant programų sistemos elgsenos ir būsenos turinius atskiriančią DCI paradigmą.

ŠALTINIAI

[AM06] A. Avram, F. Marinescu. Domain-Driven Design Quickly. InfoQ Enterprise Software Development Series, C4Media, 2006.

[BC89] K. Beck, W. Cunningham. A Labaratory For Teaching Object-Oriented Thinking. OOPSLA Conference Proceedings, New Orleans, 1989.

[BGB05] H. Baumann, P. Grassle, P. Baumann. UML 2.0 in Action: A project-based tutorial. Packt Publishing, 2005.

[Boo93] G. Booch. Object-oriented Analysis and Design with Applications. Second edition. Addison Wesley, 1993.

[CB10] J. O. Coplien, G. Bjornvig. Lean Architecture for Agile Software Development. Library of Congress Cataloging-in-Publication Data, 2010.

[Com07] COMET. Overview of the COMET methodology, 2007. [žiūrėta 2011-02-03]. Prieiga per internetą: <http://www.modelbased.net/comet/1b_Overview_html.html>

[Čap96] A. Čaplinskas. Programų sistemų inžinerijos pagrindai. I dalis. Vilnius: Matematikos ir informatikos institutas, 1996.

[Ecl94] Eclipse. OpenUP, 1994 [žiūrėta 2011-02-03]. Prieiga per internetą: <http://www.eclipse.org/epf/openup_component/openup_vision.php>

[Eva03] E. Evans. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison Wesley, 2003.

[Fow03] M. Fowler. Anemic Domain Model, 2003 [žiūrėta 2010-06-10]. Prieiga per internetą: <http://martinfowler.com/bliki/AnemicDomainModel.html>

[Ico07] ICONIX Process. Agile Without Being Extreme, 2007. [žiūrėta 2011-01-28]. Prieiga per internetą: <http://iconixprocess.com/>

[JSR06] JSR-000220 Enterprise JavaBeans 3.0, 2006 [žiūrėta 2010-06-15]. Prieiga per internetą: <http://www.jcp.org/en/jsr/detail?id=220>

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, J. Irwin. Aspect-Oriented Programming. Proceedings of Proceedings of the European Conference on Object-Oriented Programming, 1997

[MW02] A. McKean, R. Wirfs-Brock. Object Design: Roles, Responsibilities, and Collaborations. Addison Wesley, 2002.

[Obe10] R. Oberg. Contexts are the new objects, 2010 [žiūrėta 2011-01-13]. Prieiga per internetą: <http://www.jroller.com/rickard/entry/contexts_are_the_new_objects>

[Omg07] OMG Unified Modeling Language (OMG UML) Specification, Superstructure, V2.1.2. Unified Modeling Language TM, 2007.

[Par72] D.L. Parnas. On the Criteria to be Used in Decomposing Systems into Modules, Communication of the ACM, Vol.15, N°12, 1972

[Pin97] S. Pinker. How the Mind Works. Norton, New York, 1997

[Rat98] Rational Software. Rational Software White Paper, 1998 [žiūrėta 2011-02-02]. Prieiga per internetą: <http://www.ibm.com/developerworks/rational/library/content/03July/1000/1251/1251_bestpractices_TP026B.pdf>

[RC09] T. Reenskaug, J. O. Coplien. The DCI Architecture: A New Vision of Object-Oriented Programming. 2009.

[Ree01] T. Reenskaug. Modeling Systems in UML 2.0, A Proposal for a Clarified Collaboration. Mogul, Norway, 2001.

[Ree07] T. Reenskaug. Roles and Classes in Object Oriented Programming. University of Oslo, Norway, 2007.

[Ree08] T. Reenskaug. The Common Sense of Object Oriented Programming. University of Oslo, Norway, 2008.

[Ree10] T. Reenskaug. A DCI Execution Model. University of Oslo, Norway, 2010.

[Swe08] K. Swenson. Two Strategies for Handling Models: Preserving vs. Transforming. Fujitsu America, Inc. 2008.