Liet. matem. rink., **44**, *spec. nr.*, 2004, 345–350

Aspektinio programų sistemų projektavimo problemos

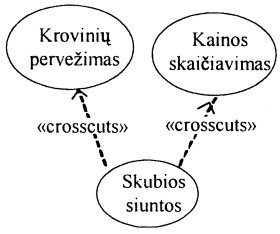
Vladislav OŠMIANSKIJ, Albertas ČAPLINSKAS* (MII)
el. paštas: vlaosm@sonexco.com, alcapl@ktl.mii.lt

Aspektinė paradigma – nauja programų sistemų inžinerijos (PSI) paradigma. Ji pradėjo formuotis nuo aspektinio programavimo, kuris buvo pasiūlytas [4] kaip viena iš moduliarizavimo technikų, padedančių programų sistemose geriau atskirti turinius. Turiniais[1] PSI vadinamos funkcinės bei nefunkcinės savybės, kurias privalo turėti kuriama programų sistema. Persikertantys turiniai – tai turiniai, kurių realizacija išbarstoma po visą sistemą, pavyzdžiui, apsauga, sinchronizacija, žurnalizavimas, ir persipina su kitų, dažniausiai, funkcinių turinių realizacija. Turinių atskirimo problema – tai gebėjimas taip moduliarizuoti sistemą, kad joje neliktų persikertančių turinių. Moduliarizavimo būdai nagrinėjami nuo pat PSI kaip savarankiškos disciplinos pradžios, tačiau, išskyrus keletą išimtinų atvejų, buvo nagrinėjamas funkcinių turinių atskirimas. Objektinėje paradigmoje turiniai yra atskiriami pasinaudojant objekto abstrakcija. Praktiniams tikslams šito pakako tol, kol programų sistemos nepasiekė tokio sudėtingumo laipsnio, kad, išbarstant nefunkcinius turinius po skirtingus sistemos modulius ir perpinant juos su funkciniais turiniais bei vienas su kitu, planuoti, įgyvendinti ir, ypač, keisti nefunkcines sistemų savybes praktiškai tapo nebeįmanoma. Sprendžiant šią problemą, praeito dešimtmečio viduryje buvo sukurta speciali programavimo technika, aspektinis programavimas, leidžianti išreikšti turinius vadinamaisiais aspektais. Kiekvieną aspektą atitinka atskiras pradinio teksto modulis. Aspektai perpinami[2] vėliau, vykdomojo kodo generavimo metu. Tam buvo pasiūlyti specialūs kompiliavimo bei kodo generavimo metodai. Pasinaudojant šiomis idėjomis, pradėta kurti aspektinės analizės, aspektinio projektavime bei aspektinio testavimo metodus ir aspektinis programavimas palaipsniui peraugo į naują PSI paradigmą, aspektinę paradigmą. Susiformavus šiai paradigmai, pradėta ieškoti atsakymų į klausimus, kaip

*Darbas atliktas Matematikos ir informatikos institute, vykdant planinę temą „Aspektinis programų sistemų projektavimas“.

[1] Kadangi lietuviškos terminijos straipsnyje nagrinėjamais klausimais kol kas nėra, pateiksime mūsų vartojamų terminų atitikmenis anglų kalba: turinys – *concern*, persikertantys turiniai – *crosscutting concerns*, turinių atskyrimas – *concern separation*, persikertančios užduotys – *crosscutting use case*, sankirtos priklausomybė – *crosscut*, aspektas – *aspect*, perpinimas – *weaving*, jungties taškas – *join point*, pjūvis – *pointcut*, rekomendacija – *advice*.

[2] Perpinimu vadinamas procesas išbarstantis aspektus po klases ir įterpiantis į tas klases aspektus realizuojančio kodo fragmentus. Taškai, kuriuose į klasės kodą terpiami aspekto kodo fragmentai, vadinami jungties taškais. Jungties taškai aprašomi vadinamaisiais pjūviais. Rekomendacijos aprašo, kaip jungties taške turi būti terpiamas aspekto kodo fragmentas (prieš nurodytą tašką, po jo, keičiant esamą kodą).



1 pav. Persikertančių turinių modeliavimas užduočių diagrama (skubios siuntos yra ir kitaip pervežamos, ir kitaip apmokestinamos).

ji dera su kitomis PSI paradigmomis, kaip ir kokiu mastu keičia koncepcinius PSI pagrindus ir kokias PSI problemas padeda spręsti geriau už kitas paradigmas. Šiame straipsnyje nagrinėjama viena iš apektinės paradigmos dedamųjų – aspektinis projektavimas. Jame analizuojami persikertančių turinių modeliavimo būdai UML[3] kalba ir nagrinėjama, kaip aspektai keičia sistemos projektavimo metodikas. Pagrindinis straipsnio tikslas – įvertinti esamą aspektinio projektavimo būklę ir identifikuoti kol kas neišspręstas problemas.

Reikalavimai, kuriuos turėtų tenkinti aspektams modeliuoti pritaikyta kalba, suformuluoti darbe [9]. Pasak autorių, kalba turėtų būti grafinė, turėti priemones specifikuoti persikertančius turinius tiek struktūros, tiek ir elgsenos požiūriu bei priemones jungties taškams aprašyti. Visos specifikacijos turėtų būti pakartotinai panaudojamos ir kombinuojamos viena su kita skirtinguose kontekstuose. Be to, detalizuojant persikirtimo specifikacijas, reikia turėti specialų atributą, kuris charakterizuotų, kokiu būdu išdetalizuotos specifikacijos elementai yra komponuojami su pjūvių elementais. Kalba taip pat turėtų leisti specifikuoti persikirtimus skirtingu granularumu ir, be abejo, būti skaidri analitikams ir projektuotojams. Pateikta keletas pasiūlymų [2, 7, 8, 9], kaip praplėsti UML kalbą taip, kad ji bent iš dalies tenkintų šiuos reikalavimus.

Darbe [7] pasiūlyta persikertančius turinius modeliuoti specialaus pobūdžio užduotimis, vadinamomis *persikertančiomis užduotimis.* UML kalba išplečiama papildomo tipo priklausomybėmis, vadinamomis *sankirtos priklausomybėmis.* Šios priklausomybės naudojamos užduočių diagramose, jomis parodoma, kurias įprastines (funkcines) užduotis gali paveikti konkreti persikertanti užduotis (1 pav.). Informaciją apie tai, kuriuose taškuose jungti funkcinius ir nefunkcinius turinius, kokius reikalavimus būtina išlaikyti juos perpinant bei apie kitus turinių perpinimo ypatumus siūloma pateikti specialiai tam skirtose *komponavimo lentelėse*[4] (2 pav.). Kiekvienai sankirtos priklausomybei sudaroma sava komponavimo lentelė. Detalizuojant užduotis sekų bei ansamblių diagramomis, kiekvienam tose diagramose vaizduojamam objektui taip yra sudaroma komponavimo lentelė. Joje nurodoma, kaip persikerta turiniai to objekto viduje.

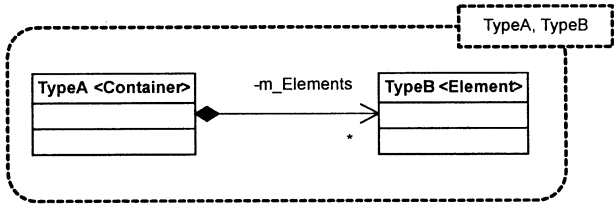
Projektuojant užduočių įgyvendinimą, turinius įgyvendinčius aspektus siūloma modeliuoti specialiomis "aspect" rūšies klasėmis. Šių klasių atributais modeliuojamos pa-

[3]UML yra Object Management Group, Inc. prekės ženklas.

[4]Angl. *composition table.*

Persikertanti užduotis: **Skubios siuntos**			
Persikerta su	*Sąlygos*	*Kompozicijos taisyklė*	*Scenarijaus žingsnis*
Krovinių pervežimas	Visada	Keičiamas žingsnis	Atvykimas pas klientą

2 pav. Užduočių komponavimo lentelė.



3 pav. Statinė struktūros diagrama, skirta perpinties struktūriniams reikalavimams modeliuoti (perpinant, keičiami tie tipų A ir B objektai, kurie yra susieti *-m_Elements* sąryšiais).

pildomas struktūros, atsirandančios išbarstant aspektus po paprastas klases, perpinant aspektų fragmentus su šių klasių kodu. Operacijomis modeliuojami bazinių klasių elgsenos pokyčiai, atsirandantys perpinant aspektus su bazinėmis klasėmis.

Vienas iš didžiausių šio persikertančių turinių modeliavimo būdo trūkumų yra reikalavimas greta grafinės UML kalbos vartoti dar ir lentelių kalbą. Kaip pašalinti šį trūkumą pasiūlyta darbe [8]. Čia perpintims aprašyti vietoje komponavimo lentelių siūloma naudoti specialias diagramas[5]. Struktūriniai perpinties reikalavimai modeliuojami statinės struktūros diagramomis (3 pav.), dinaminiai (elgsenos) – sekų ir ansamblių diagramomis (4 pav.). Perpintis modeliuojančios diagramos turi tą pačią sintaksę, kaip ir įprastos statinės struktūros, sekų bei ansamblių diagramos, tačiau jų semantika yra kita. Šiomis diagramomis nurodoma, kuriuos bazinių užduočių elementus reikia keisti, pridedant papildomą funkcionalumą.

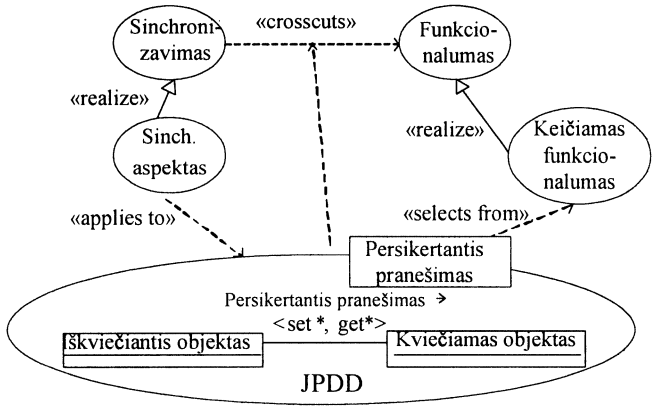
Darbe [9] autoriai siūlo išplėsti UML "advice" ir "indirectneighbour" rūšių ryšiais, skirtais persikertančių turinių poveikį kitiems modelio elementams aprašyti.

Bendras visų šių pasiūlymų [7, 8, 9] trūkumas – aspektų modeliavimas "aspect" rūšies klasėmis. Kadangi turinys gali būti išdetalizuotas iki aspektų paketo, tikslinga perpinimą modeliuoti taip pat ir paketų lygmeniu. Tam buvo pasiūlyta [2] *susiejančiojo paketo*[6] konstrukcija (5 pav.). Bazinis funkcionalumas aprašomas viename pakete, aspektai – kitame, o susiejančiojo paketo klasėmis specifikuojami pjūviai ir rekomendacijos.

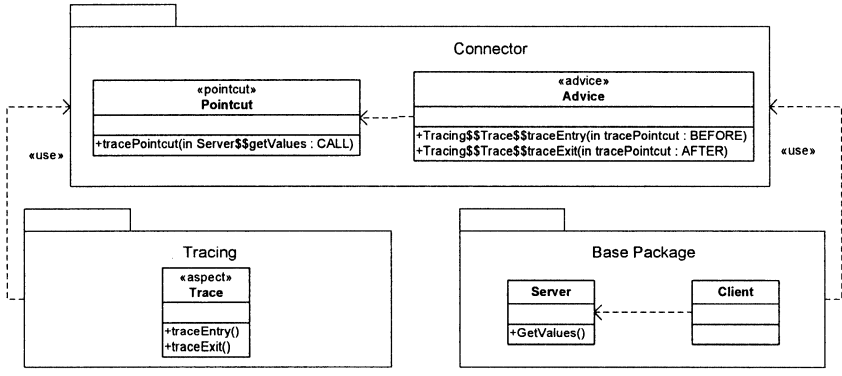
Apibendrinant apžvelgtus pasiūlymus, galima teigti, kad kol kas persikertančių turinių modeliavimo problema nėra iki galo išspręsta. Pirma, nėra iki galo aišku, kas turi būti specifikuojama, specifikuojant persikertantį turinį (aspektą), o kas – speci-

[5] Angl. *join point designation diagram* (JPDD).

[6] Angl. *connector package.*



4 pav. Ansamblio diagrama, skirta perpinties elgsenos reikalavimams modeliuoti (pridedant sinchronizavimo aspektą, keičiami visi pranešimai, prasidedantys raidėmis *set* arba *get*).



5 pav. Paketų susiejimas.

fikuojant turinių persikirtimą (perpinimą). Antra, neaišku kaip modeliuoti aspektus, keičiančius kitus aspektus. Trečia, nėra priemonių modeliuoti aspektus, kuriuos reikia perpinti dinamiškai (t.y., programos vykdymo metu). Pagaliau, kol kas galutinai nesusitarta, kaip modeliuoti vienas ar kitas persikertančių turinių savybes.

Aspektinio projektavimo metodikos paprastai yra konstruojamos modifikuojant ir plečiant objektinio projektavimo metodikas. Labiausiai išbaigti pasiūlymai pateikti darbuose [1, 5]. Darbe [1] siūloma, kaip pritaikyti užduočių modeliavimu[7] grindžiamą projektavimo metodiką aspektinės paradigmos poreikiams. Priminsime, kad šioje metodikoje projektavimas pradedamas nuo sistemos vykdomų užduočių, nagrinėjamų pačiu stambiausiu granularumo lygmeniu. Šiame lygmenyje užduotys tapatinamos su funkciniais sistemos posistemiais. Identifikavus užduotys, jos yra susiejamos priklau-

[7] Angl. *use case modelling*.

somybėmis, kiekvienai užduočiai sudaromas jos įgyvendinimo scenarijus, remiantis kuriuo projektuojamos sistemos architektūra ir jos elgsena. Norint keisti atskirų užduočių realizaciją ar papildyti sistemą naujomis užduotimis, ją tenka perprojektuoti. Aspektinė paradigma šį procesą palengvina. Darbe [1] siūloma turinius modeliuoti užduotimis. Persikertančius turinius (papildomą funkcionalumą arba esamo funkcionalumo modifikacijas) modeliuojančios užduotys su bazinėmis užduotimis susiejamos "extends" rūšies priklausomybėmis. Bazinių užduočių įgyvendinimo scenarijai keičiami, panaudojant vadinamuosius išplėsties taškus[8]. Kadangi užduotys gali būti realizuotos aspektais, išsprendžiama objektinėje paradigmoje buvusi trasavimo problema, t.y., perėjimas nuo užduočių prie pradinio kodo modulių tampa vienareikšmis. Beje, darbe [1] taip pat yra siūloma išplėsti UML modelį, įvedant specialią diagramą, skirtą specifikuoti piūvius. Neišspresta lieka vadinamoji scenarijų "sprogimo" problema. Vis pridedant ir pridedant papildomą funkcionalumą (esamo funkcionalumo modifikacijas), prigimdoma antrinių, tretinių ir žemesnių lygmenų scenarijų, ko pasėkoje suvokti visos scenarijų šeimos veikimą tampa labai sudėtinga.

Darbe [5] pasiūlyta kaip aspektinei paradigmai pritaikyti metodiką, grindžiamą ypatumų modeliavimu [5]. Ypatumais[9] šioje metodikoje vadinami savarankiški funkciniai turiniai. Ypatumai yra grupuojami ir lokalizuojami komponentuose. Kiekvienas komponentas yra kuriamas (projektuojamas, programuojamas, testuojamas ir t.t.) atskirai. Darbe [5] pasiūlyta šią metodiką išplėsti, įvedant ypatumus, modeliuojančius persikertančius turinius. Šie ypatumai turi būti realizuojami aspektais, perpinamais su įprastus ypatumus realizuojančių klasių kodu. Kadangi nefunkciniai turiniai gali būti prieštaringi, tai, projektuojant sistemą, tenka spręsti jų konfliktus. Darbe [5] pasiūlyta konfliktų sprendimo strategijas aprašyti aspektais ir šitaip palengvinti jų modifikavimą.

Taigi, apibendrinant aukščiau atliktą apžvalgą, matosi, kad aspektinis projektavimas kol kas tebėra mokslinių tyrimų stadijoje. Nors iš esmės jau yra pakankamai aišku, kaip projektuoti panaudojant aspektus, nei projektavimo strategijos, nei projektavimo procedūros dar nėra pakankamai brandžios. Iki galo nėra išspręstos ir persikertančių turinių bei aspektų modeliavimo UML kalba problemos, nesukurta visuotinai pripažinta UML notacija aspektams projektuoti.

Literatūra

1. I. Jacobson, Use cases and aspects working seamlessly together, *Journal of Object Technology*, **2**(4), 7–28 (2003).
2. I. Groher, T. Baumgarth, Aspect-orientation from design to code, in: *Early Aspects 2004: Aspect-Oriented Requirements Engineering and Architecture Design, Workshop Papers, AOSD 2004*, Lancaster (2004). `http://trese.cs.utwente.nl/workshops/early-aspects-2004/workshop_papers.htm`
3. Hannemann, G. Kiczales, Design pattern implementation in Java and spectJ, in: *Proceedings of the 17th Annual ACM conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November (2002), pp. 161–173.

[8] Angl. *extension point*.
[9] Angl. *feature*.

4. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin, Aspect-oriented programming, in: *ECOOP '97 Conference Proceedings, LNCS 1241*, Springer-Verlag (1997), pp. 220–242.
5. S.R. Palmer, J.M. Felsing, *A Practical Guide to Feature-Driven Development.* Prentice Hall PTR (2002).
6. J. Pang, L. Blair, Refining feature driven development – a methodology for early aspects, in: *Early Aspects 2004: Aspect-Oriented Requirements Engineering and Architecture Design, Workshop Papers, AOSD 2004*, Lancaster (2004).
`http://trese.cs.utwente.nl/workshops/early-aspects-2004/workshop_papers.htm`.
7. G. Sousa, S. Soares, P. Borba, J. Castro, Separation of crosscutting concerns from requirements to design: adapting an use case driven approach, in: *Early Aspects 2004: Aspect-Oriented Requirements Engineering and Architecture Design, Workshop Papers, AOSD 2004*, Lancaster (2004).
`http://trese.cs.utwente.nl/workshops/early-aspects-2004/workshop_papers.htm`.
8. D. Stein, S. Hanenberg, R. Unland, Modeling pointcuts, in: *Early Aspects 2004: Aspect-Oriented Requirements Engineering and Architecture Design, Workshop Papers, AOSD 2004*, Lancaster (2004).
`http://trese.cs.utwente.nl/workshops/early-aspects-2004/workshop_papers.htm`.
9. D. Stein, S. Hanenberg, R. Unland, Position paper on aspect-oriented modeling: issues on representing crosscutting features, in: *AOSD-UML: Aspect-Oriented Modeling with UML, Workshop Papers*, AOSD (2003).
`http://lglwww.epfl.ch/workshops/aosd2003/papers/Schedule.htm`.

SUMMARY

V. Ošmianskij, A. Čaplinskas. Problems in aspect oriented design (AOD)

The paper aims to evaluate the state of the art of AOD. It discusses main approaches to aspect modelling with UML, investigates the impact of aspects on OO design methods and identifies open issues of AOD.

Keywords: aspect, design, crosscutting concern, join point, pointcut, feature, use case, UML.