VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
PROGRAMŲ SISTEMŲ BAKALAURO STUDIJŲ PROGRAMA

Java kodo skirstymo į paketus strategijos

Java Packaging Strategies

Kursinis darbas

Atliko: 4 kurso 1 grupės studentė
Agota Šuliokaitė (parašas)

Darbo vadovas: lekt. Gediminas Rimša (parašas)

Vilnius – 2021

TURINYS

Įvadas

Temos aktualumas. Objektinio stiliaus programų sistemų kūrimo pasaulyje itin pabrėžiama moduliarumo svarba. Moduliarizacija, arba dar kitaip klasteriavimu, dekompozicija, gali būti vadinamas ir programų sistemų skirstymas į paketus [ADS+09] [BT04] [SBB+05]. Kodo moduliarizavimas jau seniai žinomas kaip vienas iš būdų pagerinti lankstumą, aiškumą ir įgalinti trumpesnį projekto vystymo laiką [Par72]. Kokybiškai apibrėžta ir įgyvendinta modulinė struktūra pagerina architektūrinį stabilumą, palaikomumą, testuojamumą ir perpanaudojamumą, kas natūraliai veda link ilgalaikiškesnio ir taupesnio sistemos kūrimo [ADS+09].

Klasės, paketai, vardų sritys, komponentai, posistemės - visos šios koncepcijos gali būti laikomos moduliais. Šiame darbe moduliarizacija yra laikomas susijusių klasių grupavimas į paketus. Programų sistemų kūrimo istorijoje yra nemažai terminų ir šiek tiek skirtingų klasių grupavimo apibrėžimų [ADS+09], tačiau dažniausiai vartojami terminai yra paketai ir vardų sritys. Reikia pastebėti, kad dauguma šiame darbe apžvelgiamos ir aptariamos teorijos gali būti pritaikoma įvairioms modulinėms sistemoms - ne tik paketams kaip klasių grupavimo struktūrai.

Pagal šio darbo pavadinimą, tyrimo laukas turėtų būti dar labiau susiaurintas - iki konkretaus ir gerai žinomo Java paketų ir subpaketų koncepto. Tačiau kaip ir modulio apibrėžimo atveju, daug programų sistemų moduliarizavimo mokslinių tyrimų taikomi ne vien tik Java paketams, tad specialūs pastebėjimai dėl Java specifikos bus pateikiami tik jei moksliniame straipsnyje pateikiamos metodikos negalėtų būti tiesiogiai pritaikomos Java programavimo kalbai.

Šiame darbe naudojama paketų struktūrizavimo strategijos sąvoka nusako gaires ir taisykles, kuriomis remiantis sistemos kodas gali būti skirstomas į paketus.

Juergen Hoeller, Spring karkaso bendraautorius, savo kalboje „The Spring Experience“ konferencijoje [Hoe07] pastebėjo, kad „paketų struktūra yra stebėtinai nelengva pabaisa“. Klasių organizavimas į paketus yra viena iš labiausiai apleistų kodo organizavimo ir architektūrinio tvarkingumo problemų. Dauguma projektų prasideda nuo nedidelės kodo bazės ir vėliau, kai projektas plečiasi ir išryškėja pasirinktos paketų struktūros minusai, dažniausiai jau būna per vėlu iš esmės restruktūrizuoti kodą ir retas programuotojas ryžtasi pertvarkyti visą kodo bazę (kuri veikia!). J. Hoeller pastebi, kad kodo skirstymo į paketus užduoties nepalengvina ir tas faktas, kad Java kalbos kūrėjai nusprendė nepateikti griežtų paketo koncepto naudojimo rekomendacijų. Kadangi paketai yra apibrėžti netiesiogiai, didelės apimties programų kūrėjai dažnai susiduria su tinkamos paketų struktūros kūrimo ir palaikymo problemomis [Hau02].

Kadangi programų sistemų moduliarumo tema diskusijos vyksta jau ne vieną dešimtmetį, galima susidaryti įspūdį, kad yra viena ar dvi paketų struktūrizavimo strategijos, kurių veikimas yra patvirtintas kaip geriausias dauguma atvejų. Iš tikrųjų, yra nuomonių, kad sistemos turėtų būti projektuojamos naudojant vienodus metodus, tačiau skirtingų metodų šalininkai vis dar negali sutarti, kuris yra geriausias bendruoju atveju. Vis dėlto daug tyrinėtojų pažymi, kad vieno geriausio sprendimo požiūris yra iš esmės ydingas. Skirtingi architektūriniai stiliai turi skirtingas stiprybes ir silpnybes ir kiekvienu atveju turi būti pasirenkamas konkrečiai situacijai pritaikytas sprendimas. Skirtingos kodo skirtymo į paketus strategijos yra neišvengiamos [Sha95]. Remiantis šia problema, formuluojamas šio **kursinio darbo tikslas**: pasiūlyti kriterijų rinkinį, leidžiantį objektyviai įvertinti

ir palyginti skirtingas objektinio stiliaus programų sistemų skirstymo į paketus strategijas.

Keliami uždaviniai tikslui pasiekti:

- apžvelgti esamus mokslinius tyrimus objektinio stiliaus programų sistemų moduliarizavimo tema;
- apžvelgti siūlomus sistemų moduliarizavimo kriterijus;
- pasiūlyti kriterijų rinkinį, skirtą palyginti skirtingas strategijas;
- pritaikyti kriterijų rinkinį vertinant kelias pasirinktas strategijas.

1. Moduliarumo problema

Programų sistemų architektūros projektavimas apima kuriamos sistemos skirstymą į modulius, modulių tarpusavio sąveiką, atsakomybių priskyrimą moduliams ir modulių plečiamumą [BT04]. Skirstymo į paketus procesas yra klasifikuojamas kaip kombinatorinė optimizacijos problema. Paketo lygio sistemos matą galima naudoti kaip šios problemos tinkamumo funkciją. Tarp programų sistemų architektų labiausiai paplitęs metodas yra apibrėžti tinkamumo funkciją, kuri maksimaliai padidina atskirų paketų sanglaudą ir maksimaliai sumažina sankibą tarp visų paketų [EA15].

Objektinio stiliaus programų sistemų kūrimo kontekste paketai sudaro esminius modulinės struktūros komponentus. E. Hautus straipsnyje „Improving Java Software Through Package Structure Analysis“ [Hau02] teigia, kad Java programavimo kalba suteikia galimybę dekomponuoti kuriamas sistemas į modulius naudojantis paketo konstruktu. Kiekviena Java klasė yra paketo dalis, o paketai yra hierarchiškai struktūrizuojami paketų medyje. Todėl paketai Java programose yra laikomi moduliais. Pasak autoriaus, paketai yra ypatingai svarbūs, nes jie yra tinkamiausi aukšto lygio Java programų architektūrai apibrėžti. Taip pat manoma, kad paketų kokybė daro įtaką sistemos stabilumui, tad paketai gali būti naudojami ir sistemos stabilumo įvertinimui [EA19].

Taigi kuriant naują sistemą svarbu įvertinti, kokius esminius kriterijus turėtų atitikti pasirinkta sistemos moduliarizavimo strategija ir pasirinkti geriausiai tinkantį sprendimą.

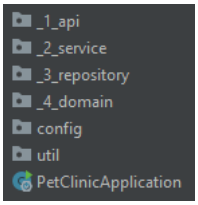
2. Strategijos

Kokie yra pirmieji žingsniai, padedantys pagrindą bendrai sistemos formai? Tai paketų struktūra. Paketų struktūra yra pats pirmas dalykas, su kuriuo susiduria programuotojas. Tai - vienas svarbiausių sistemos kodo aspektų [Sys18]. Šiame skyriuje pristatomos dviejų pagrindinių kodo skirtymo į paketus strategijų šeimos - sluoksninė ir komponentinė. Kiekvienos strategijos variacijų yra ne viena, tačiau šiame darbe bus nagrinėjami bendriausi variantai, autorės nuomone geriausiai atspindintys visos šeimos savybes.

Daugelis mokslinių šaltinių ir internetinių straipsnių nepateikia konkrečių taisyklių, aiškiai apibrėžiančių, kaip reikia spręsti praktines ir dažnai pasitaikančias problemas naudojant vieną ar kitą strategiją. Galima rasti nemažai nuomonių ir argumentų, tačiau nėra griežtai nusistovėjusių taisyklių, tad paketų struktūrizavimo strategijos yra laikomos daugiau gairėmis nei konkrečių taisyklių rinkiniais.

2.1. Sluoksninė strategija

Remiantis sluoksnine strategija, sistema dalinama horizontaliai - atskiriami techniniai sluoksniai, atliekantys skirtingas architektūrines funkcijas [Sys18]. Kiekvienas aukšto lygio paketas atitinka architektūrinį sistemos sluoksnį. Priklausomybės tarp sluoksnių yra vienakryptės ir eina nuo išorinių interfeisų (angl. *application programming interface*, toliau - API) iki duomenų saugojimo (angl. *persistence*) lygmens.



1 pav. Sluoksniai

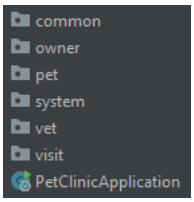
Sluoksninė strategija yra populiariausia numatytoji strategija, sutinkama tiek demonstraciniuose mokomuosiuose pavyzdžiuose, tiek realiose sistemose. Tokį sėkmingą sluoksninės architektūros įsitvirtinimą galimai lėmė tai, kad ši strategija yra vienas iš seniausiai apibrėžtų architektūrinių stilių [SM09].

Verta pastebėti, kad sluoksninė strategija išreikštinai neapibrėžia, kokiam sluoksniui reikia priskirti tokius paketus kaip *config* ar *util* - ar konfigūracines ir pagalbines klases reikia išskirstyti po sluoksnius, kuriuose jos yra naudojamos, ar geriau šias klases laikyti atskirai ne sluoksniniuose paketuose. Šis ir kiti konkretūs klausimai yra palikti spręsti sistemų kūrėjams individualiai be jokių bendrai apibrėžtų ir visuotinai priimtų taisyklių.

2.2. Komponentinė strategija

Remiantis komponentine strategija, sistema dalinama vertikaliai - atskiriami skirtingą funkcionalumą įgyvendinantys dalykinės srities komponentai [Sys18]. Skirtingose situacijose kompo-

nentai gali būti suskirstyti skirtingai, nes, kitaip nei atskiriant architektūrinius sluoksnius, dalykinės srities komponentus galima apibrėžti nevienareikšmiškai. Komponentas gali būti suprantamas kaip vienas atskiras panaudos atvejis, tačiau komponentu gali būti ir panaudos atvejų rinkinys, susijęs su konkrečia dalykinės srities problema.



2 pav. Komponentai

Kiekviename pakete yra visi failai, susiję su konkrečiu funkcionalumu:

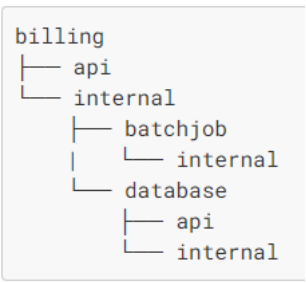
3 pav. Pet paketas

Komponentinė strategija į pirmą planą iškelia dalykinės srities funkcionalumą.

2.2.1. Vidinė loginio paketo struktūra

Pasirinkus komponentinę strategiją dažnai kyla klausimas, kaip komponentų paketus toliau efektyviai skaidyti į subpaketus.

Tom Hombergs [Hom18] pastebi, kad Java kalbos siūlomas matomumo modifikatorių funkcionalumas nėra pakankamas, nes jis riboja paketų skirstymą į subpaketus. Norint, kad du vieno paketo subpaketai galėtų naudoti vienas kito klases, jas reikia paversti *public*, o tai duoda prieigą prie klasės ne tik kitiems subpaketams, bet visai sistemai. T. Hombergs siūlo naudoti *api-internal* vidinę subpaketų struktūrą, kuri leidžia praplėsti Java matomumo modifikatorių siūlomą funkcionalumą ir lengviau valdyti priklausomybes tarp paketų naudojant tokius įrankius kaip ArchUnit. Atsiranda galimybė lengvai kurti dideles hierarchines komponentų struktūras, aiškiai apribojant kiekvieno paketo matomumo lygį ir priklausomybes.



4 pav. *api-internal* subpaketų struktūra [Hom18]

3. Paketų strategijų vertinimo kriterijai

Viena svarbiausių programų sistemos savybių yra palaikomumas. Palaikomumas pagal IEEE programinės įrangos terminologijų žodyną [Com+90] apibrėžiamas kaip matas, parodantis, kaip lengva pakeisti sistemą ar sistemos komponentą taisant klaidas, gerinant veikimą ar kitus atributus, prisitaikant prie besikeičiančių reikalavimų ir aplinkos. Lengvai palaikomos sistemos reikalauja mažiau sąnaudų atsirandant naujiems reikalavimams, taisant klaidas ir palaikant sistemos kokybę. Kaip kurti sistemas, kurias būtų lengva palaikyti, išlaikyti jų kokybę laikui bėgant ir atsirandant vis naujiems pakeitimams?

Nors subjektyvus vertinimas kodo peržiūrų metu vis dar yra labai svarbi sistemų palaikomumo vertinimo dalis [Wel01], tiek kodo peržiūrų metu, tiek renkantis bendrą sistemos kodo organizavimo strategiją sistemos palaikomumo atžvilgiu naudinga naudoti susistemintus vertinimo kriterijus.

Sistemos palaikomumą galima įvertinti naudojant keletą kriterijų, susijusių su sistemos aiškumu ir keičiamumu [Lan02]:

- sankiba,
- sanglauda,
- stabilumas,
- kodo dubliavimas - DRY principo laikymasis,
- kodo aiškumas - KISS principo laikymasis,
- apimtis,
- atvirumo-uždarumo principas,
- susiejimas su dalykinės srities reikalavimais.

Papildomai šiame darbe vertinama paketų strategijų įtaka Java matomumo modifikatorių naudojimui.

3.1. Sanglauda ir sankiba

Sankibą galima apibrėžti kaip priklausomybių tarp atskirų komponentų skaičių. Sanglaudą galima vadinti sankiba, kuri yra neišvengiama dėl dalykinės srities specifikos ir pasireiškia kaip priklausomybių komponento viduje skaičius. Pagal R. C. Martin sanglaudos principus [Mar00], kartu paketuose esančios klasės turi būti kartu naudojamos ir kartu besikeičiančios. Kitaip tariant, kartu esančios klasės turi būti glaudžiai susijusios dalykinės srities atžvilgiu. Laikantis šio principo palengvėja pokyčių įgyvendinimas sistemoje (mažėja vienam funkcionalumui reikalingų įgyvendinti ir besikeičiančių paketų skaičius).

Martin Fowler straipsnyje „Reducing Coupling“ [Fow01] pažymi, kad sankiba, kartu su sanglauda, yra vienas iš seniausiai apibrėžtų programų sistemų dizaino rodiklių. Jeigu keičiant vieną

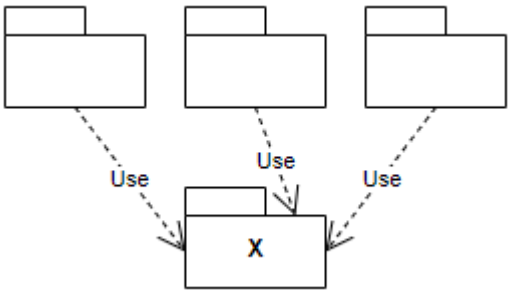
kodo dalį būtinai reikia pakeisti ir kitą kodo dalį - tarp šių kodo vietų egzistuoja sankiba. Sankiba taip pat atsiranda, kai vienas komponentas naudoja kitame komponente esantį kodą.

Siekiamybė. Pasak Juerger Hoeller [Hoe07], pagrindinės norimos paketų charakteristikos yra žema sankiba su kitais paketais ir didelė sanglauda tarp klasių (ir vidinių paketų) paketo viduje. Sankiba yra neišvengiama moduliarizavimo pasėkmė, tačiau sankibą reikia stengtis minimizuoti ir valdyti atsižvelgiant ne tik į kiekybinius sankibos vertinimo rezultatus, bet ir į kitus ryšių tarp komponentų aspektus, aprašomus tolimesniuose šio darbo skyreliuose.

3.2. Stabilumas

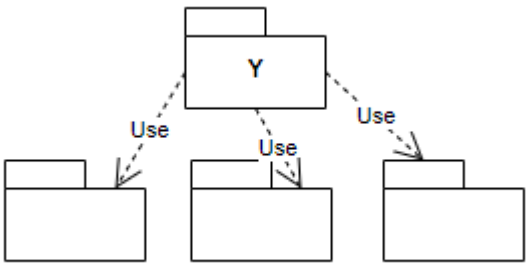
R. C. Martin straipsnyje „Design Principles“ [Mar00] aprašo stabilių priklausomybių principą (angl. *Stable Dependencies Principle - SDP*).

Stabilumas - tai darbo, reikalingo pakeisti objekto būseną, kiekis. Svarbus faktorius, darantis įtaką paketo stabilumui, yra nuo jo priklausančių paketų skaičius. Norint atlikti pakeitimą tokiame pakete, pakeitimai dažnai turi būti atlikti visuose nuo jo priklausančiuose paketuose.



5 pav. Stabilus paketas (pagal R. C. Martin „Design Principles“)

5 pav. vaizduojamas stabilus paketas X. Jis turi tris priežastis nesikeisti - tris nuo jo priklausančius paketus. Tačiau pats paketas X nepriklauso nuo jokio kito paketo.



6 pav. Nestabilus paketas

6 pav. vaizduojamas nestabilus paketas Y. Jis neturi jokių kitų nuo jo priklausančių paketų, tačiau pats paketas Y priklauso nuo trijų kitų paketų. Pokyčiai bet kuriame iš trijų paketų privers keistis ir paketą Y.

Stabilumo matas. Paketo stabilumas gali būti matuojamas trimis paprastais matais:

- *Ca* Įeinančios priklausomybės. Skaičius klasių paketo išorėje, kurios priklauso nuo klasių paketo viduje.
- *Ce* Išeinančios priklausomybės. Skaičius klasių paketo išorėje, nuo kurių priklauso klasės paketo viduje.
- *I* Nestabilumas. $I = \frac{Ce}{(Ca + Ce)}$. [0, 1]

Stabilus paketas: *I = 0*, nėra nė vienos išeinančios priklausomybės. Nestabilus paketas: *I = 1*, nėra nė vienos įeinančios priklausomybės.

Siekiamybė. Kadangi viena svarbiausių gerai struktūrizuoto kodo savybių yra tai, kaip lengva jį keisti, komponentų stabilumas ne visada yra pageidautinas. Kuo paketas nestabilesnis, tuo lengviau jame atlikti pakeitimus. Svarbiausia užtikrinti, kad priklausomybės nebūtų nevaldomos ir būtų laikomasi R. C. Martin rekomenduojamos stabilumo krypties taisyklės: paketai turi priklausyti nuo tų paketų, kurių *I* matas yra mažesnis. Kai kurie šaltiniai [AMT13] atkreipia dėmesį, kad sistemos komponentų stabilumas yra ypatingai svarbus renkantis perpanaudojamus komponentus iš kitų šaltinių ir įkomponuojant juos kuriamame projekte. Tad praktikoje svarbu atsirinkti, kurios programos dalys turi būti lengvai keičiamos, o kurios turi būti stabilios ir keičiamos kuo mažiau.

3.3. DRY ir KISS principai

DRY (angl. *Don't Repeat Yourself*) principas - besikartojantis kodas turi keistis kartu, o sužiūrėti sunku, tad atsiranda klaidų. M. Fowler ragina vadovautis „Trejeto taisykle“ [Fow18] nusprendžiant, kada kodas turėtų būti perpanaudojamas. Jeigu kodas pakartojamas trijose vietose - jį reikia perpanaudoti.

Sistemos kodas, kuris itin smarkiai atitinka DRY principą, dažnai pažeidžia kitą svarbų principą - KISS (angl. *Keep It Simple Stupid*), kuris iš esmės padeda kodą išlaikyti lengvai skaitomu ir suprantamu. Perpanaudojamas kodas dažnai būna atskirtas nuo vietų, kuriose jis yra naudojamas (perkeliant jį į atskirus metodus, klases ar paketus). Tokiu atveju sistemos programuotojams dažnai reikia perskaityti ir susipažinti ne su viena sistemos vieta, kad suprastų, kaip veikia vienas konkretus funkcionalumas.

Per smarkiai taikomas DRY principas vieno funkcionalumo kodą išskaido po keletą sistemos vietų, o tai sumažina sistemos paprastumą ir aiškumą, kurį skatina KISS principas, taip pat dažnu atveju padidina sistemos komponentų sankibą. Kaip teigiama Abel Avram straipsnyje „Using DRY: Between Code Duplication and High-Coupling“ [Avr12], sistemų kūrimo principų negalima taikyti po vieną, būtina kiekvienoje situacijoje įvertinti galimas principo taikymo pasėkmes ir derinti su kitais žinomais principais.

Siekiamybė. Šie abu principai turėtų būti derinami tiek tarpusavyje, tiek ir su kitais principais, kad nebūtų pasiekti tokie kraštutinumai, kai DRY principas lemia didesnę sankibą tarp sistemos komponentų ir didesnį bendrą sistemos sudėtingumą. Vis dėlto, Philipp Hauer [Hau20b] pastebi, kad KISS principas yra svarbesnis už DRY, todėl iškilus dilemai, kuriam principui teikti pirmenybę, aiškumas ir suprantamumas turėtų eiti pirma. Galų gale, tiek DRY, tiek KISS, tiek ir

kiti programuotojams žinomi principai gali pavirsti anti-principais, jeigu yra taikomi atskirai vienas nuo kito ir neatsižvelgiant į konkrečią situaciją.

3.4. Apimtis

Paketų dydis, klasių, metodų, kodo eilučių skaičius - sistemos apimtį galima matuoti pasitelkiant skirtingus matavimo vienetus.

Siekiamybė. Gerosios praktikos yra atskirų sistemos elementų apimtį išlaikyti kuo mažesnę. Įvairūs autoriai [Mar08; McC04] pabrėžia tiek metodų, tiek klasių, tiek paketų, tiek sistemų apimties limitų svarbą. Nors griežti limitai (tokie, kaip ne daugiau 7 metodų vienoje klasėje) dažnai sukelia dirbtinų taisyklių pojūtį, vienokie ar kitokie nusistatyti limitai paskatina programuotojus būti sąmoningesniais ir greičiau pastebėti, kada kodo komponentas nebeatlieka vienos nedalomos funkcijos ir gali būti suskaidomas į kelis mažesnius ir lengviau suprantamus komponentus. Jim Bird [Bir13] teigia, kad paprastos ir aiškios taisyklės yra geriau įsimenamos ir padeda programuotojams aktyviai palaikyti kodo kokybę.

3.5. Atvirumo-uždarumo principas

Pagal atvirumo-uždarumo (angl. *open-closed principle*) principą, sistemos elementai turi būti atviri praplėtimui, bet uždari modifikavimui, t.y. bendru atveju naujas sistemos funkcionalumas turi būti pridedamas nemodifikuojant jau esančio funkcionalumo.

Siekiamybė. Nors šis principas yra apibrėžtas klasių lygmenyje, jį galima taikyti ir renkantis paketų struktūros dizainą. Paketų struktūra turi leisti pridėti naują funkcionalumą iš esmės nemodifikuojant egzistuojančių paketų.

3.6. Dalykinės srities atspindėjimas sistemoje

Robert C. Martin įvardina sąvoką „klykianti architektūra“ [Mar11]. Kaip pastato architektūrinis brėžinys su kambarių planais „klykia“ apie tai, kokios paskirties yra projektuojamas pastatas, taip sistemos architektūra turėtų „klykti“ apie įgyvendinamą dalykinę sritį, o ne iš kokių „plytų“ - techninių elementų - ji sudaryta.

Robert C. Martin klausia: „tai ką gi jūsų sistemos architektūra klykia? Kai pasižiūrite į aukščiausio lygio paketų struktūrą, į klases aukščiausiuose paketų lygmenyse, ar jie klykia: Ligoninės sistema, Buhalterijos sistema ar Inventoriaus Valdymo sistema? Ar jie klykia Rails, Spring, Hibernate, ASP?“

Kodo organizavimas neturi būti priklausomas nuo techninių karkasų ir sprendimų - tai tik įrankiai sprendimo įgyvendinimui. Dalykinės srities panudos atvejai yra pagrindinė sprendžiama problema, todėl kodo struktūra turi būtent juos ir atspindėti. Gera sistemos architektūra skaitytojams turėtų „klykti“ apie pačią sistemą, o ne apie naudojamus techninius sprendimus.

Klykiančios architektūros koncepcija susijusi su sistemos komponentų susiejimu su dalykinės srities reikalavimų įgyvendinimu. Projektuojant ir įgyvendinant programų sistemas vienas svarbiausių dalykų yra sistemos reikalavimų ir juos įgyvendinančių sistemos komponentų atsekamu-

mas ir susiejimas. Šiame darbe bus vertinama, kaip skirtingos strategijos prisideda prie reikalavimų susiejimo. Susiejimą lengviau pastebėti ir užtikrinti, jeigu kiekvienas komponentas įgyvendina uždarą susijusių reikalavimų rinkinį.

Siekiamybė. Apie sistemos kodo susiejimo su dalykine sritimi svarbą kalba ir Eric Evans knygoje „Domain Driven Design“ [EE04]. Evans teigia, kad dalykinės srities kalba yra vienas svarbiausių dalykų komunikacijoje tarp sistemos kūrėjų ir kitų su interesuotų šalių, tad yra labai svarbu, kad sistemos kodas - pagrindinis sistemos produktas - kuo aiškiau atspindėtų dalykinės srities kalbą ir naudojamas esybes.

3.7. Java matomumo modifikatorių naudojimas

Java kalba siūlo keturis elementų (klasių, laukų, metodų) matomumo modifikatorius. Numatytasis (*package-private*) modifikatorius prieigą suteikia tik to paties paketo elementams (subpaketų elementai nepriklauso paketams). Apsaugotasis (*protected*) modifikatorius suteikia prieigą to paties paketo elementams ir visoms išvestinėms klasėms. Privatus (*private*) modifikatorius prieigą suteikia tik tos pačios klasės elementams. Viešas (*public*) modifikatorius suteikia prieigą visiems sistemos elementams.

Kadangi matomumo modifikatoriai yra glaudžiai susiję su paketų ribomis, paketų struktūra daro didelę įtaką, kaip modifikatoriai bus naudojami sistemos kode.

Siekiamybė. Geroji praktika yra kodo elementams parinkti kiek įmanoma mažesnį matomumo lygį, kad būtų išvengta perteklinių ir nevaldomų priklausomybių, kai kiekvienas sistemos elementas gali pasiekti ir naudoti visus kitus elementus.

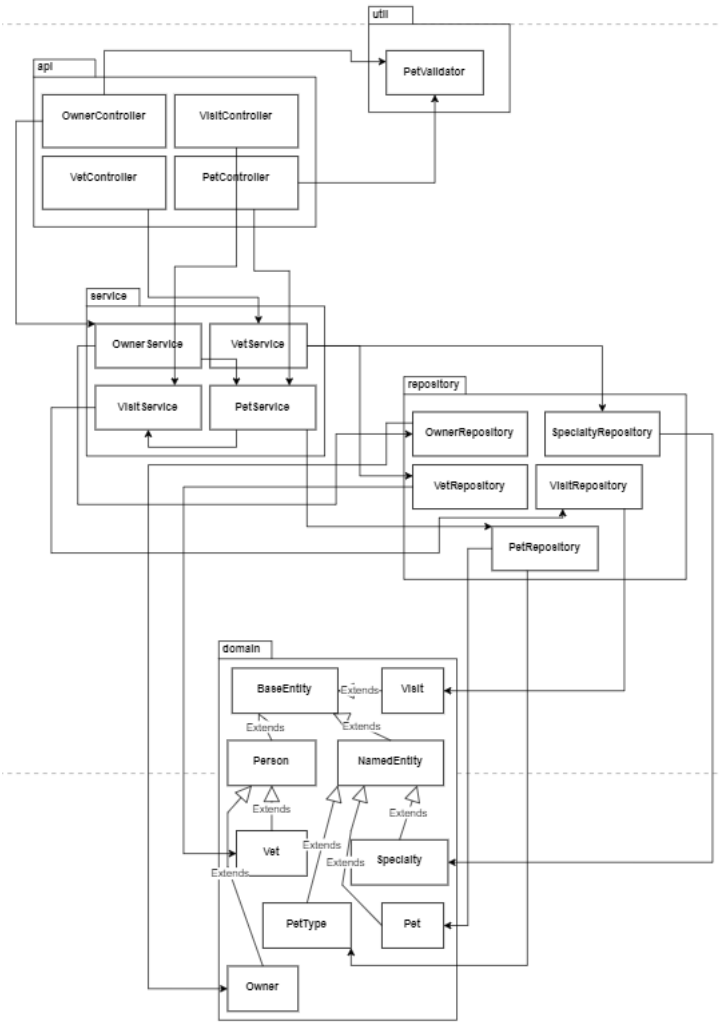
4. Sluoksninės ir komponentinės strategijų palyginimas

4.1. Kriterijai

Autorė siūlo objektinio stiliaus sistemų skirstymo į paketus strategijas vertinti ir lyginti tarpusavyje pagal sistemos palaikomumo kriterijus ir keletą papildomų atributų:

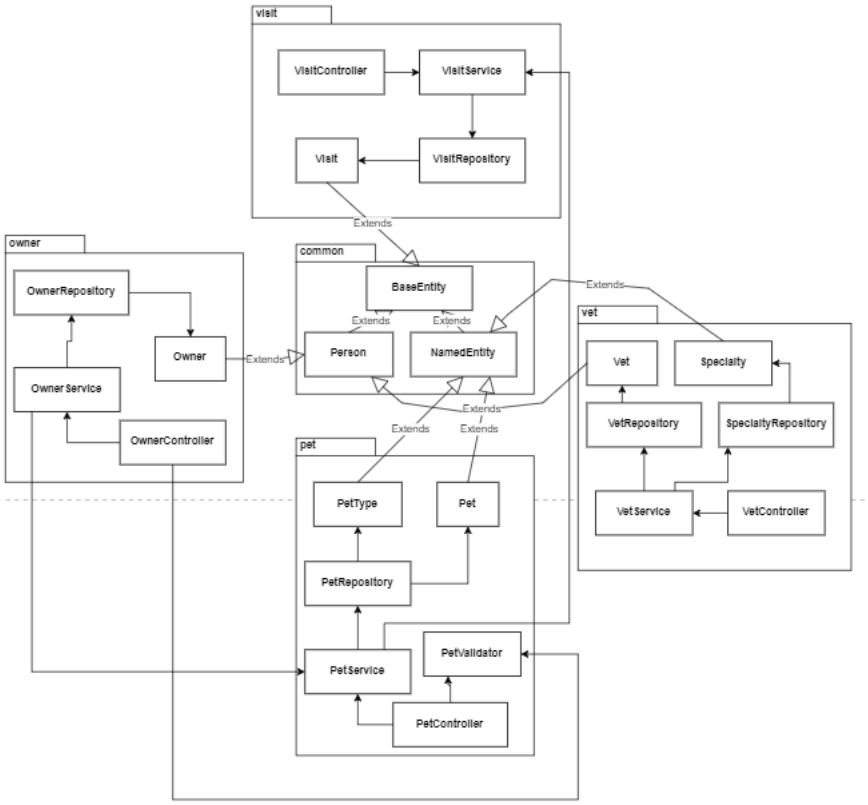
- **Strategijos principas.** Pagrindinė idėja, kuria remiasi strategija.
- **Java matomumo modifikatoriai.** Ar strategija išnaudoja Java siūlomų matomumo modifikatorių naudą?

Šiame skyriuje siekiama išbandyti, ar siūlomi strategijų palyginimo kriterijai yra praktiškai pritaikomi. Šiam tikslui pasiekti naudojamos dvi *Spring PetClinic*[1] versijos: sluoksninė versija[2] ir komponentinė versija[3].



7 pav. *PetClinic* struktūra naudojant sluoksninę strategiją

[1] https://projects.spring.io/spring-petclinic/
[2] https://github.com/agosu/petclinic-layers
[3] https://github.com/agosu/petclinic-components



8 pav. *PetClinic* struktūra naudojant komponentinę strategiją

4.2. Strategijos principas

Sluoksninė strategija. Paketais atskiriami techniniai architektūriniai sluoksniai. Sistema skaidoma horizontaliai. Pateiktame pavyzdyje yra išskiriami *api*, *service*, *repository* ir *domain* sluoksniai.

Komponentinė strategija. Paketais atskiriami dalykinės srities funkcionalumą įgyvendinantys komponentai. Sistema skaidoma vertikaliai. Techniniai sluoksniai atskiriami klasių lygmenyje. Pateiktame pavyzdyje yra išskiriami *owner*, *pet*, *vet*, *visit* komponentai.

Išvada. Strategijos principas parodo esminį skirtumą tarp dviejų nagrinėjamų strategijų. Bendruoju atveju keliais sakiniais aprašant strategijos esmę galima išskirti ne tik esminius skirtumus tarp lyginamų strategijų, bet ir egzistuojančius panašumus.

4.3. Sankiba

Vertinimas. Jeigu sankibą tarp paketų apibrėžiame kaip tarppaketinių priklausomybių skaičių, tokiu atveju ją galima įvertinti vienareikšmiškai - kiekybiškai. Kuo daugiau tarppaketinių priklausomybių, tuo didesnė sankiba tarp paketų. Vienas iš tiesiogiai kode sankibą atspindinčių dalykų yra *import* sakinių skaičius. Visos naudojamos klasės, kurios nepriklauso tam pačiam paketui, turi būti importuotos naudojant *import* sakinį. Kuo daugiau priklausomybių nuo kitų paketų klasių - tuo daugiau *import* sakinių. Skaičiuojant bus įtraukiami tik tie *import* sakiniai, kurie importuoja kitas projekto klases (bet ne Java ar Spring Framework klases), nes šiame darbe nagrinėjami būtent

ryšiai tarp projekto klasių. Kadangi projektiniai pavyzdžiai nėra dideli, skaičiavimai buvo atlinkti rankiniu būdu peržiūrint klases.

Sluoksninė strategija. Sankiba tarp paketų yra itin didelė, nes kiekvienas dalykinės srities funkcionalumas įgyvendinamas visuose sluoksniuose. Kiekvieno paketo sankiba su kitais paketais yra pateikta 1 lentelėje.

1 lentelė. Priklausomybių skaičius tarp paketų naudojant sluoksninę strategiją

api-service	api-repository	api-domain	api-util	service-repository
4	0	0	2	5
service-domain	service-util	repository-domain	repository-util	domain-util
0	0	6	0	0

Pateikti skaičiai atspindi, kad naudojant sluoksninę strategiją priklausomybės susidaro tik tarp gretimų sluoksnių komponentų. Jeigu sluoksniai nėra šalia vienas kito, jie tiesioginių priklausomybių neturi ir 1 lentelėje jų skaičius yra 0.

2 lentelėje pateikiamas bendras visų priklausomybių skaičius naudojant sluoksninę strategiją ir vidutinis *import* sakinių skaičius kiekvienoje klasėje.

2 lentelė. Sankibos naudojant sluoksninę strategiją santrauka

bendras priklausomybių skaičius	17
vidutinis *import* sakinių skaičius klasėje	1,4

Komponentinė strategija. Sankiba tarp paketų yra mažesnė, nes paketai neprivalo bendrauti tarpusavyje, kad įgyvendintų reikiamą dalykinės srities funkcionalumą. Klasės, įgyvendinančios vieną funkcionalumą, yra viename pakete. Susijusių funkcionalumų paketai pariklausomybių greičiausiai turės, bet jų nebus tarp visų paketų. Kiekvieno paketo sankiba su kitais paketais yra pateikta 3 lentelėje.

3 lentelė. Priklausomybių skaičius tarp paketų naudojant komponentinę strategiją

owner-pet	owner-vet	owner-visit	owner-common	pet-vet
2	0	0	1	0
pet-visit	pet-common	vet-visit	vet-common	visit-common
1	2	0	2	1

4 lentelėje pateikiamas bendras visų priklausomybių skaičius naudojant komponentinę strategiją ir vidutinis *import* sakinių skaičius kiekvienoje klasėje.

4 lentelė. Sankibos naudojant komponentinę strategiją santrauka

bendras priklausomybių skaičius	9
vidutinis *import* sakinių skaičius klasėje	0,7

Išvada. Naudojant komponentinę strategiją sankiba tarp paketų yra mažesnė, nei naudojant sluoksninę strategiją.

4.4. Sanglauda

Vertinimas. Kaip ir sankibą, sanglaudą galima išmatuoti kiekybiškai, skaičiuojant vidines priklausomybes tarp komponento sudedamųjų dalių (šio darbo atveju tarp pakete esančių klasių ir subpaketų). Moksliai šaltiniai šia tema [GS09] siūlo ne tik skaičiuoti priklausomybes, bet remtis ir tokiais matais kaip metodų, klasių vidinių subpaketų panašumas, kuris taip pat gali būti skirstomas į funkcinį, duomenų ir laiko panašumą [SLL99].

Kadangi naudojami *PetClinic* pavyzdžiai nėra pakankamai kompleksiški, šiame darbe sanglauda bus vertinama tik skaičiuojant klasines priklausomybes paketo viduje, kurios parodo, ar pakete įgyvendinamas funkcionalumas yra pakankamai susijęs, t.y. ar paketą galima laikyti funkciniu dalykinės srities komponentu. Skaičiavimai atlikti rankiniu būdu peržiūrint klases.

Sluoksninė strategija. Sanglauda yra nedidelė, nes viename sluoksnyje yra klasės, susijusios su skirtingais dalykinės srities funkcionalumais. Kiekvieno paketo sanglauda su kitais paketais yra pateikta 5 lentelėje.

5 lentelė. Priklausomybių skaičius paketų viduje naudojant sluoksninę strategiją

api	service	repository	domain	util
0	2	0	8	0

Bendras priklausomybių paketų viduje skaičius - 10.

Kadangi sluoksninė strategija skaido sistemą horizontaliai pagal techninius sluoksnius, paketų sanglauda iš dalykinės srities perspektyvos natūraliai gaunasi labai nedidelė.

Komponentinė strategija. Sanglauda tarp paketų yra itin stipri, nes visos klasės ir subpaketai įgyvendina vieną konkretų dalykinės srities funkcionalumą. Kiekvieno paketo sanglauda su kitais paketais yra pateikta 6 lentelėje.

6 lentelė. Priklausomybių skaičius paketų viduje naudojant komponentinę strategiją

owner	pet	visit	common	vet
3	5	3	2	5

Bendras priklausomybių paketų viduje skaičius - 18.

Kadangi komponentinė strategija skaido sistemą būtent pagal dalykinės srities komponentus, paketų sanglauda natūraliai gaunasi labai didelė.

Išvada. Naudojant komponentinę strategiją sanglauda paketų viduje yra didesnė, nei naudojant sluoksninę strategiją.

4.5. Stabilumas

Vertinimas. Šiame darbe stabilumas vertinamas kiekybiškai - skaičiuojant įeinančias ir išeinančias kiekvieno paketo priklausomybes. Skaičiavimai atlikti rankiniu būdu peržiūrint klases.

Sluoksninė strategija. 7 lentelėje pateikiami kiekvieno *PetClinic* paketo stabilumo matai (įeinančios priklausomybės *Ca*, išeinančios priklausomybės *Ce* ir nestabilumo lygmuo *I*).

7 lentelė. Paketų stabilumas naudojant sluoksninę strategiją

	api	service	repository	domain	util
Ca	0	4	5	6	2
Ce	6	5	6	0	0
I	1	0,55	0,54	0	0

Komponentinė strategija. 8 lentelėje pateikiami kiekvieno *PetClinic* paketo stabilumo matai.

8 lentelė. Paketų stabilumas naudojant komponentinę strategiją

	owner	pet	visit	common	vet
Ca	0	2	1	5	0
Ce	3	3	1	0	2
I	1	0,6	0,5	0	1

Išvada. Naudojant sluoksninę strategiją paketai yra stabilesni, nes nuo jų priklauso daugiau kitų paketų. Naudojant komponentinę strategiją paketai mažiau stabilūs, nes yra labiau nepriklausomi vienas nuo kito. Kuris variantas yra siektinas, reikia vertinti kiekvienoje situacijoje atskirai, atsižvelgiant į ansktesniame skyriuje aprašytą stabilumo kryptį ir kitus kriterijus.

4.6. Principai

Vertinimas. Šiame darbe nagrinėjamus tris principus - DRY, KISS ir atvirumo-uždarumo - įvertinti vienareikšmiškai nėra lengva. Naudojamas *PetClinic* pavyzdys nėra pakankamai kompleksiškas, kad juo remiantis galima būtų vertinti strategijų ir principų santykį, tad bus naudojamas ekspertinis vertinimas, paremtas teoriniais šaltiniais.

Praktiškai vertinant kodo atitikimą DRY principui, dažnai padeda integruotų programavimo aplinkų (angl. *Integrated Development Environment*) įskiepiai, automatiškai pažymintys kodo blokus, kurie kartojasi keletą kartų.

Praktiškai įvertinti kodo atitikimą KISS principui padeda kodo peržiūros, kurių metu parašytą kodą skaito ir bando suprasti kodo nerašę asmenys.

4.6.1. DRY principas

Sluoksninė strategija. Pasak Tom Hauer [Hau20a], sluoksninė strategija skatina DRY principo laikymąsi bendrai naudojamas (sluoksnio) klases (kodo elementus) laikant viename pakete ir generalizuojant kodą, kad jį galima būtų perpanaudoti daugelyje kitų (sluoksnio) klasių.

Komponentinė strategija. Didesnis kodo dublikavimas ar didesnis panašaus kodo sistemoje kiekis gali būti šios strategijos naudojimo pasėkmė. Kadangi strategija iš esmės skatina kūrėjus kiekvieną paketinį komponentą išlaikyti kuo labiau nepriklausomą vieną nuo kito, atsiradus panašaus funkcionalumo poreikiui keliuose komponentuose, dažniau linkstama tą funkcionalumą dalinai dublikuoti pritaikant konkrečiai kiekvienam komponentui.

Išvada. Nors DRY principo laikymasis iš esmės priklauso nuo žemesnių implementacijos detalių, nei paketai, pasirinkta paketų strategija gali paskatinti programuotojus priimti daugiau ar mažiau DRY principą atitinkančius sprendimus.

4.6.2. KISS principas

Sluoksninė strategija. Sunkiau pastebėti konkrečias vietas, kurios yra svarbios vienam ar kitam konkrečiam funkcionalumui. Bendrai naudojamos klasės reikalauja iš programuotojo daugiau ar mažiau suprasti visą sistemą, kad jis būtų tikras, jog keičiant vieną bendrinės klasės vietą, nebus paveiktos kitos sistemos dalys. Šiuo atveju KISS principas yra pažeidžiamas [Hau20a].

Komponentinė strategija. Nors ši strategija skatina rašyti daugiau panašaus pasikarotjančio kodo, mažesnis abstraktumo lygis dažnai yra daug geriau suprantamas. Modifikuojant egzistuojantį funkcionalumą programuotojui nebereikia blaškytis tarp daugelio abstrakčių klasių, kurios apsunkina modifikuojamam funkcionalumui aktualių vietų radimą.

Išvada. Kaip ir DRY principo atveju, KISS principo laikymąsi daugiausiai nulemia žemesnio lygio įgyvendinimo detalės, tačiau pasirinkta paketų strategija gali daryti įtaką aukšto lygio sistemos aiškumui ir nulemti programuotojų požiūrį į tam tikras įgyvendinimo detales.

4.6.3. Atvirumo-uždarumo principas

Sluoksninė strategija. Funkcionalumo keitimas arba trynimas lemia pokyčius didelėje dalyje sistemos paketų. Ši strategija inkapsuliuoja techninius sluoksnius. Tai yra naudinga, jeigu keičiasi konkreti techninė implementacija (pavyzdžiui, pereinama nuo vienos ORM (angl. *Object Relational Mapping*) sistemos prie kitos, arba pasikeičia karkasas). *PetClinic* pavyzdyje atsiradus reikalavimui valdyti klinikas, neišvengiamai būtų paveikti visi egzistuojantys paketai nuo *Clinic-Controller api* pakete iki *Clinic domain* pakete. Norint įgyvendinti naujus reikalavimus, *PetClinic* sistema privalo būti atvira modifikavimui - taip pažeidžiant atvirumo-uždarumo principą paketų lygmenyje.

Komponentinė strategija. Ši strategija inkapsuliuoja dalykinės srities funkcionalumus. Tai yra naudinga, jeigu verslo reikalavimai dažnai keičiasi ir juos reikia dažnai įgyvendinti. *PetClinic* pavyzdyje atsiradus reikalavimui valdyti klinikas, atsirastų naujas *Clinic* paketas su visomis naujomis susijusiomis klasėmis. Taip naujas funkcionalumas į sistemą įvestas būtų iš esmės ne keičiant egzistuojančius paketus, bet pridedant naują. Aišku, priklausomai nuo poreikio ir implementacijos, gali tekti minimaliai keisti kitų paketų kodą, pavyzdžiui, iš *PetService* kviečiant *ClinicService*, tačiau komponentinės strategijos atveju atvirumo-uždarumo principas pažeidžiamas gerokai mažiau.

Išvada. Abi strategijos palaiko atvirumo-uždarumo principą, tačiau skirtinguose koncepciniuose lygmenyse. Naudojant sluoksninę strategiją lengviau inkapsuliuoti ir praplėsti, techninius sistemos elementus. Naudojant komponentinę strategiją, lengviau plečiami funkciniai komponentai, pridedant naują dalykinės srities funkcionalumą įgyvendinančius paketus. Kadangi dalykinės srities reikalavimai paprastai keičiasi dažniau, nei techniniai įgyvendinimo sprendimai, komponentinės strategijos suteikiamas lansktumas pridedant naujus funkcinius komponentus yra svarbesnis.

4.7. Apimtis

Vertinimas. Sistemos elementų, šiuo atveju - paketų, apimtį galima įvertinti vienareikšmiškai. Šiame darbe sistemos apimtis bus matuojama paketų kiekiu ir klasių bei subpaketų kiekiu

paketuose.

Sluoksninė strategija. Plečiantis sistemai didėja paketų ir subpaketų apimtis. Pavyzdžiui, atsiradus naujam *PetClinic* sistemos reikalavimui valdyti klinikas - priskirti joms veterinarus, valdyti patalpų užimtumą, įrangą, kiekviename sluoksnyje atsirastų bent po vieną naują klasę (*ClinicController*, *ClinicService*, *ClinicRepository*, *Clinic* etc.). Visos *Clinic*<> klasės būtų neišvengiamai susijusios tarpusavyje, dar labiau padidėtų paketų sankiba.

Komponentinė strategija. Plečiantis sistemai, atsiranda daugiau komponentinių paketų. Kaip teigia Sandi Metz [Met16], kodo komponentai turi būti kuo mažesni ir žinoti kuo mažiau apie vienas kitą. Atsiradus naujam *PetClinic* sistemos reikalavimui valdyti klinikas, atsirastų naujas *Clinic* paketas, kuriame būtų visos susijusios klasės. Jau egzistuojantys paketai nesiplėstų, išskyrus nebent *common* paketą, jeigu atsirastų bendra logika su kitais jau egzistuojančiais paketais.

Išvada. Sluoksninė strategija skatina sistemą plėstis į gylį - didėja klasių ir subpaketų skaičius paketuose. Komponentinė strategija lemia sistemos plėtimąsi į plotį - didėja komponentinių paketų skaičius. Bendru atveju naviguoti po sistemos paketus yra lengviau, kai jie yra mažesni (atitinkamai, kaip ir navigacijos po klases ir metodus atvejais).

4.8. Dalykinės srities atspindėjimas sistemoje

Vertinimas. Šis sistemos atributas bus vertinamas remiantis tiesioginiu pavyzdžių nagrinėjimu ir ekspertiniais autorės pastebėjimais.

Sluoksninė strategija. Paketų vardai informuoja naudotoją apie architektūrinį sistemos dizainą. Daugumoje Java sistemų jis yra standartinis ir visiem įprastas. Aukšto lygio paketų vardai nesisieja su dalykinės srities vardais. *PetClinic* pavyzdyje sistema atspindi ne pagrindines dalykinės srities esybes, o architektūrinius sluoksnius: *api, services, repository, domain*.

Komponentinė strategija. J. Hoeller teigia [Hoe07], kad komponentiniai paketai padeda sukurti natūralesnius paketų vardus, palengvinančius navigavimą po kodą. Paketų vardai suteikia vertingą informaciją apie dalykinę sritį - kokios yra pagrindinės sistemos funkcijos, kokios yra pagrindinės dalykinės srities esybės, kuriomis yra manipuliuojama. Kadangi dalykinių sričių yra gerokai daugiau, nei populiariausių Java sistemų architektūrinių sprendimų, sistemos kūrėjams ir palaikytojams sistemoje paketų vardais dokumentuota dalykinės srities informacija yra gerokai naudingesnė ir reikalingesnė. *PetClinic* pavyzdyje sistema atspindi pagrindines dalykinės srities esybes *pet, vet, owner, visit*.

Išvada. Kadangi pokyčiai dažniausiai kyla iš dalykinės srities konteksto, paketų struktūra, kuri aiškiai atspindi šį kontekstą, leidžia pakeitimus atlikti greičiau. Paketai turėtų atspindėti ne techninius, bet dalykinės srities kontekstus [Bos17].

4.9. Java matomumo modifikatoriai

Vertinimas. Matomumo modifikatorių naudojimą galima įvertinti kiekybiškai - skaičiuojant kiekvieno modifikatoriaus panaudojimo sistemoje skaičių. Skaičiavimai atlikti naudojant *IntelliJ IDE* paiešką.

Sluoksninė strategija. Dėl didelės sankibos tarp paketų, įprastai naudojamas *public* modifikatorius, taip neišnaudojant *private* ir *package-private* modifikatorių siūlomų galimybių. *PetClinic* pavyzdyje visos klasės ir didžioji dalis klasių metodų yra *public*. 9 lentelėje pateikiami visų modifikatorių pasikartojimo atvejai.

9 lentelė. Modifikatorių panaudojimas sluoksninėje strategijoje

public	package	protected	private
98	0	0	28

Komponentinė strategija. Pagrindinis yra numatytasis *package-private* modifikatorius ir *public* modifikatorius naudojamas tik kai to tikrai reikalauja implementacija (pavyzdžiui, naudojant pagalbinio paketo funkcijas). *import* sakiniai tokiu atveju aiškiai parodo klasės priklausomybes nuo kitų paketų. Vis dėlto, esant didesnei sistemai ir naudojant subpaketus, *package-private* modifikatorius taip pat nėra išnaudojamas (subpaketo klasės turi būti *public*, kad tėvinis paketas galėtų jomis naudotis). Šiai problemai spręsti galima pasitelkti tokius sprendimus kaip *api-internal* vidiniai paketai [Hom18]. 10 lentelėje pateikiami visų modifikatorių pasikartojimo atvejai *PetClinic* pavyzdyje:

10 lentelė. Modifikatorių panaudojimas naudojant komponentinę strategiją

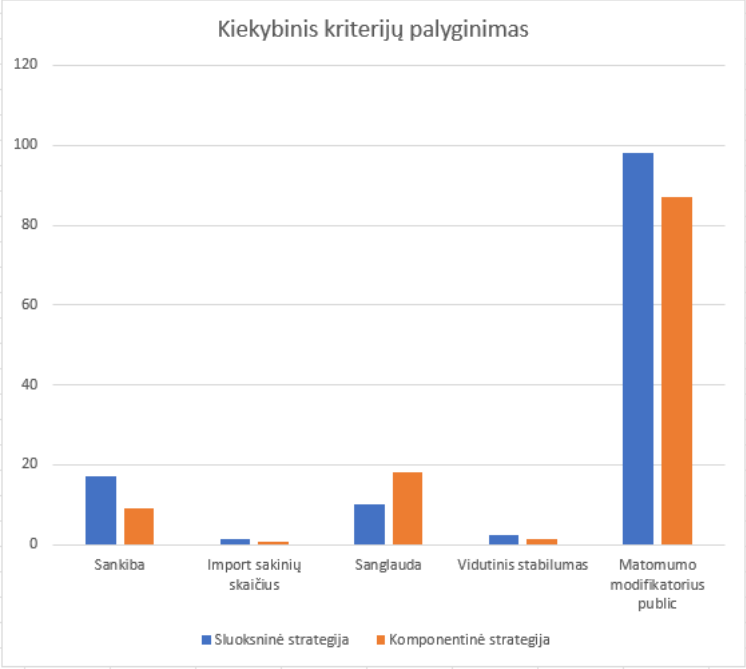
public	package	protected	private
87	11	0	28

Išvada. Nors *PetClinic* pavyzdys nėra didelis, net ir tokio dydžio projekte matosi, kad naudojant komponentinę strategiją kai kuriuos *public* modifikatorius galima pakeisti *package* modifikatoriais.

5. Apibendrinimas

Apibendrinant galima teigti, kad iš esmės visi nagrinėti kodo skirstymo į paketus strategijų vertinimo kriterijai yra glaudžiai susiję su sankiba ir sanglauda. Tiek paketų stabilumas, tiek DRY, KISS, atvirumo-uždarumo principai, tiek susiejimas su dalykinės srities reikalavimais, tiek Java *import* sakinių bei matomumo modifikatorių naudojimas vienu ar kitu aspektu veikia sistemos komponentų sankibą ir sanglaudą. 9 pav. diagramoje vaizduojamas kiekybiškai išmatuojamų kriterijų palyginimas. Išskyrus sanglaudą, kuri iš esmės yra atvirkščias dydis sankibai, likusių kriterijų palyginimas parodo, kad esant didesnei sankibai, didesnis yra ir *import* sakinių skaičius, vidutinis paketo stabilumas ir *public* matomumo modifikatoriaus panaudojimų skaičius. Nors laikoma, kad kuo mažesnė sankiba - tai universali siekiamybė, tokių sudėtingesnių su sankiba susijusų kriterijų, kaip stabilumas, siektina reikšmė skirtingose situacijose gali skirtis ir detaliau šiame darbe nėra nagrinėjama.

Galima daryti išvadą, kad sankibą ir sanglaudą reikia vertinti ne tik kaip išorinių ir vidinių priklausomybių kiekį - tai yra esminis sistemos kodo skirstymo į paketus įvertinimo kriterijus, kuris nėra ir negali būti vienareikšmiškai įvertinamas. Sistemoje egzistuojančias priklausomybes svarbu vertinti naudojant įvairius aspektus, leidžiančius visapusiškai vertinti sankibą ir sanglaudą, kadangi skirtingose architektūrinėse situacijose ir tarp skirtingų komponentų sankiba nėra lygiavertė - būtina įvertinti ne tik kiekybinį, bet ir kokybinį dėmenį.



9 pav. Kiekybinis kriterijų palyginimas

Rašydama šį kursinį darbą autorė priėjo prie išvados, kad pirmas žingsnis vertinant sistemos skaidymo į paketus strategiją yra įvertinti paketų sankibą ir sanglaudą, nes priklausomybių tarp sistemos komponentų kiekis ir pobūdis yra itin svarbus bendrame sistemos kūrimo ir palaikymo kontekste.

Bandant įvertinti dvi pasirinktas strategijas remiantis keletu populiarių sistemų kūrimo prin-

cipų buvo padaryta išvada, kad joks principas negali būti taikomas beatodairiškai pagal apibrėžimą, neatsižvelgianti į kitus principus. Kitu atveju, jeigu principai yra taikomi atskirai vienas nuo kito, jų įgyvendinimas dažnai gali būti prieštaringas vienas kito atžvilgiu (kaip yra DRY ir KISS principų atveju). Į bendrą taikomų principų visumą ir derėjimą tarpusavyje reikia atsižvelgti ir vertinant paketų strategijas.

Kuriant programų sistemas yra svarbu įvertinti konkrečią techninę situaciją bei dalykinės srities reikalavimus. Šiame darbe pasiūlyti, nagrinėti ir praktiškai įvertinti objektinio stiliaus sistemų moduliarizavimo naudojant paketus kriterijai nedidele dalimi gali palengvinti sistemų kūrėjų sprendimą renkantis paketų strategiją, geriausiai atitinkančią poreikius ir reikalavimus. Tolimesniems darbams paliekama detaliau atsakyti į klausimus, kaip praktiškai panaudoti ir derinti tarpusavyje apžvelgtus kriterijus vertinant ir renkantis kodo skirstymo į paketus strategijas.

Rezultatai ir išvados

Rezultatai:

- naudojant *Spring Petclinic* koncepciją sukurti du pavyzdiniai projektai, naudojantys skirtingas kodo skirstymo į paketus strategijas;
- pasiūlytas skirstymo į paketus strategijų vertinimo kriterijų rinkinys;
- kriterijų rinkinys praktiškai ir teoriškai pritaikytas vertinant sukurtus pavyzdinius projektus.

Išvados:

- nors kodo skirstymo į paketus tema yra aktuali ir plačiai nagrinėjama tiek moksliniuose darbuose, tiek internetiniuose šaltiniuose, trūksta konkrečiai aprašytų pavyzdžių, išreikštinai apibrėžiančių strategijų taikymą praktiškai sudėtingesnėse realiose sistemose;
- paketų struktūrai įvertinti yra naudojama daug skirtingų kiekybinių ir kokybinių kriterijų, tačiau detaliau nagrinėjant jų pagrindimą ir veikimą paaiškėjo, kad didžioji dalis kriterijų glaudžiai siejasi su sistemos komponentų sankiba ir sanglauda, taigi būtent šie du atributai yra pagrindinis sistemos vertinimo kriterijus, kurį kiti kriterijai papildo;
- dažniausiai naudojama sluoksninė paketų struktūrizavimo strategija nebūtinai yra geriausia - daug autoritetingų šaltinių pateikia svarių argumentų, kodėl komponentinė sistemos moduliarizavimo strategija yra naudingesnė sistemos palaikomumo atžvilgiu nei sluoksninė alternatyva.

Šaltiniai

[ADS+09] Hani Abdeen, Stéphane Ducasse, Houari Sahraoui ir Ilham Alloui. Automatic package coupling and cycle minimization. *2009 16th Working Conference on Reverse Engineering*, p. 103–112. IEEE, 2009. Prieiga per internetą: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.650.6228\&rep=rep1&type=pdf [žiūrėta 2021-09-11].

[AMT13] Lerina Aversano, Marco Molfetta ir Maria Tortorella. Evaluating architecture stability of software projects. *2013 20th Working Conference on Reverse Engineering (WCRE)*, p. 417–424. IEEE, 2013. Prieiga per internetą: https://www.researchgate.net/publication/257609545_Evaluating_Architecture_Stability_of_Software_Projects [žiūrėta 2021-12-04].

[Avr12] Abel Avram. Using DRY: Between Code Duplication and High-Coupling. https://www.infoq.com/news/2012/05/DRY-code-duplication-coupling/, 2012. [žiūrėta 2021-12-04].

[Bir13] Jim Bird. Rule of 30 – When is a Method, Class or Subsystem Too Big? https://dzone.com/articles/rule-30-\%E2\%80\%93-when-method-class-or, 2013. [žiūrėta 2021-12-04].

[Bos17] Robert Bosch. Happy Packaging. https://javadevguy.wordpress.com/2017/12/18/happy-packaging/, 2017. [žiūrėta 2021-09-23].

[BT04] Markus Bauer ir Mircea Trifu. Architecture-aware adaptive clustering of OO systems. *Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings.* P. 3–14. IEEE, 2004. Prieiga per internetą: https://ieeexplore.ieee.org/abstract/document/1281401 [žiūrėta 2021-09-11].

[Com+90] IEEE Standards Coordinating Committee ir k.t. IEEE standard glossary of software engineering terminology (IEEE Std 610.12-1990). Los Alamitos. *CA: IEEE Computer Society*, 169:132, 1990. Prieiga per internetą: http://www.informatik.htw-dresden.de/~hauptman/SEI/IEEE_Standard_Glossary_of_Software_Engineering_Terminology.pdf [žiūrėta 2021-11-30].

[EA15] Shouki A Ebad ir Moataz A Ahmed. Functionality-based software packaging using sequence diagrams. *Software Quality Journal*, 23(3):453–481, 2015. Prieiga per internetą: https://link.springer.com/article/10.1007/s11219-014-9245-3 [žiūrėta 2021-09-21].

[EA19] Shouki A Ebad ir Moataz Ahmed. Investigating the effect of software packaging on modular structure stability. *Computer Systems Science and Engineering*, 34(5):283–296, 2019. Prieiga per internetą: https://www.researchgate.net/profile/Shouki-Ebad/publication/348654540_Investigating_the_Effect_of_Software_Packaging_on_Modular_Structure_Stability/links/

60153539299bf1b33e35360e/Investigating-the-Effect-of-Software-Packaging-on-Modular-Structure-Stability.pdf [žiūrėta 2021-09-20].

[EE04] Eric Evans ir Eric J Evans. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004. Prieiga per internetą: https://books.google.lt/books?id=hHBf4YxMnWMC [žiūrėta 2021-12-01].

[Fow01] Martin Fowler. Reducing coupling. *IEEE Software*, 18(4):102–104, 2001. Prieiga per internetą: https://martinfowler.com/ieeeSoftware/coupling.pdf [žiūrėta 2021-09-18].

[Fow18] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018. Prieiga per internetą: https://ieeexplore.ieee.org/abstract/document/8278488 [žiūrėta 2021-10-04].

[GS09] Gui Gui ir Paul D Scott. Measuring Software Component Reusability by Coupling and Cohesion Metrics. *J. Comput.*, 4(9):797–805, 2009. Prieiga per internetą: http://www.jcomputers.us/vol4/jcp0409-01.pdf [žiūrėta 2021-12-04].

[Hau02] Edwin Hautus. Improving Java software through package structure analysis. *International Conference Software Engineering and Applications*, 2002. Prieiga per internetą: https://ehautus.home.xs4all.nl/papers/PASTA.pdf [žiūrėta 2021-09-11].

[Hau20a] Philipp Hauer. Package by feature. https://phauer.com/2020/package-by-feature/, 2020. [žiūrėta 2021-10-02].

[Hau20b] Philipp Hauer. The Wall of Coding Wisdoms in Our Office. https://phauer.com/2020/wall-coding-wisdoms-quotes/, 2020. [žiūrėta 2021-12-04].

[Hoe07] Juergen Hoeler. Code organization guidelines for lange codebases. https://www.infoq.com/presentations/code-organization-large-projects/, 2007. [žiūrėta 2021-10-19].

[Hom18] Tom Hombergs. Clean Architecture Boundaries with Spring Boot and ArchUnit. https://reflectoring.io/java-components-clean-boundaries/, 2018. [žiūrėta 2021-10-17].

[Lan02] Rikard Land. Measurements of software maintainability. *Proceedings of the 4th ARTES Graduate Student Conference*, p. 1–7, 2002. Prieiga per internetą: http://www.artes.uu.se/events/gsconf02/papers/Land_Maintainability.pdf [žiūrėta 2021-11-30].

[Mar00] Robert C Martin. Design principles and design patterns. *Object Mentor*, 1(34):597, 2000. Prieiga per internetą: https://fi.ort.edu.uy/innovaportal/file/2032/1/design_principles.pdf [žiūrėta 2021-09-18].

[Mar08] R.C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Robert C. Martin Series. Pearson Education, 2008. ISBN: 9780136083252. Prieiga per internetą: https://books.google.lt/books?id=_i6bDeoCQzsC [žiūrėta 2021-12-04].

[Mar11] Robert C. Martin. Screaming architecture. https://blog.cleancoder.com/uncle-bob/2011/09/30/Screaming-Architecture.html, 2011. [žiūrėta 2021-10-18].

[McC04] S. McConnell. *Code Complete*. Developer Best Practices Series. Microsoft Press, 2004. ISBN: 9780735619678. Prieiga per internetą: https://books.google.lt/books?id=QnghAQAAIAAJ [žiūrėta 2021-12-04].

[Met16] Sandi Metz. The wrong abstraction. https://sandimetz.com/blog/2016/1/20/the-wrong-abstraction, 2016. [žiūrėta 2021-10-18].

[Par72] David L Parnas. On the criteria to be used in decomposing systems into modules. *Pioneers and Their Contributions to Software Engineering*, p. 479–498. Springer, 1972. Prieiga per internetą: https://link.springer.com/chapter/10.1007/978-3-642-48354-7_20 [žiūrėta 2021-09-11].

[SBB+05] Olaf Seng, Markus Bauer, Matthias Biehl ir Gert Pache. Search-based improvement of subsystem decompositions. *Proceedings of the 7th annual conference on Genetic and evolutionary computation*, p. 1045–1051, 2005. Prieiga per internetą: https://dl.acm.org/doi/abs/10.1145/1068009.1068186 [žiūrėta 2021-09-21].

[Sha95] Mary Shaw. Architectural issues in software reuse: It's not just the functionality, it's the packaging. *Proceedings of the 1995 Symposium on Software reusability*, p. 3–6, 1995. Prieiga per internetą: http://www.cs.cmu.edu/afs/cs.cmu.edu/project/vit/ftp/pdf/Packaging.pdf [žiūrėta 2021-09-11].

[Sys18] Hirondelle Systems. Package by feature, not layer. http://www.javapractices.com/topic/TopicAction.do?Id=205, 2018. [žiūrėta 2021-09-22].

[SLL99] Frank Simon, Silvio Loffler ir Claus Lewerentz. Distance based cohesion measuring. *proceedings of the 2nd European Software Measurement Conference (FESMA)*, tom. 99, 1999. Prieiga per internetą: https://www.researchgate.net/profile/Claus-Lewerentz/publication/2445809_Distance_Based_Cohesion_Measuring/links/0046351c3dc037a9ab000000/Distance-Based-Cohesion-Measuring.pdf [žiūrėta 2021-12-04].

[SM09] Juha Savolainen ir Varvana Myllarniemi. Layered architecture revisited—Comparison of research and practice. *2009 Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture*, p. 317–320. IEEE, 2009. Prieiga per internetą: https://www.researchgate.net/publication/224605935_Layered_Architecture_Revisited_-_Comparison_of_Research_and_Practice [žiūrėta 2021-12-01].

[Wel01] Kurt D Welker. The software maintainability index revisited. *CrossTalk*, 14:18–21, 2001. Prieiga per internetą: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.444.9945&rep=rep1&type=pdf [žiūrėta 2021-11-30].

Santrumpos ir sąvokų apibrėžimai

- Sankiba (angl. *coupling*)
- Sanglauda (angl. *cohesion*)
- Tinkamumas (angl. *fitness*)
- Panaudos atvejis (angl. *user case*)
- Kodo bazė (angl. *code base*)
- KISS (angl. *Keep It Simple Stupid*)
- DRY (angl. *Don't Repeat Yourself*)
- SDP (angl. *Stable Dependency Principle*)
- OCP (angl. *Open Closed Principle*)