VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
PROGRAMŲ SISTEMŲ BAKALAURO STUDIJŲ PROGRAMA

Kodo skirstymo į paketus šablonų tyrimas

Analysis of code packaging patterns

Bakalauro baigiamasis darbas

Atliko: Martyna Ubartaitė

Darbo vadovas: Gediminas Rimša

Darbo recenzentas: doc. dr. Audronė Lupeikienė

Vilnius – 2024

Santrauka

Glaustai aprašomas darbo turinys: pristatoma nagrinėta problema ir padarytos išvados. Santraukos apimtis ne didesnė nei 0,5 puslapio. Santraukų gale nurodomi darbo raktiniai žodžiai. Automatiškai naudojamos lietuviškos kabutės: „tekstas“.

Raktiniai žodžiai: raktinis žodis 1, raktinis žodis 2, raktinis žodis 3, raktinis žodis 4, raktinis žodis 5

Summary

Santrauka anglų kalba. Santraukos apimtis ne didesnė nei 0,5 puslapio. Automatiškai naudojamos angliškos kabutės: “tekstas”.

Keywords: keyword 1, keyword 2, keyword 3, keyword 4, keyword 5

Turinys

Įvadas

Teisingai įgyvendintas kompiuterinės sistemos dizainas yra vienas iš kritinių sėkmingo verslo elementų. Tam, jog informacines sistemas naudojantis verslas išlaikytų stabilų augimą, yra būtina sukurti sistemą, kuri sumažintų atotrūkį tarp organizacijos tikslų ir jų įgyvendinimo galimybių. Mąstant apie programinio kodo dizainą, kodo paketų kūrimas, klasių priskyrimas jiems ir paketų hierarchijos sudarymas paprastai nėra pagrindinis prioritetas, tačiau tai parodo praleistą galimybę padaryti sistemos dizainą labiau patikimu [Sho19], suprantamu [Eli10] ir lengviau palaikomu. Modernios sistemos yra didžiulės, programinis kodas yra padalintas į daugybę failų, kurie išskaidyti per skirtingo gylio direktorijas, todėl apgalvotai išskirstytas programinis kodas daro daug didesnę įtaką kodo kokybei, nei gali atrodyti iš pirmo žvilgsnio. Sistemos paketų struktūros analizė norint įvertinti programinės įrangos kokybę tampa vis svarbesne tema dėl augančio failų ir paketų skaičiaus [Eli10].

Norint išsiaiškinti, kaip programinis kodas gali būti skirstomas efektyviausiai, tam jog jo struktūra darytų teigiamą įtaką sistemos kokybei, reikalinga atlikti skirstymo į paketus šablonų analizę – išsiaiškinti galimus šablonus, kaip skirstyti programinį kodą į paketus, turėti aiškius šablonų apibrėžimus su jų privalumais bei trūkumais. Šiame darbe minint *šabloną kodo skirstymui į paketus* turima omenyje taisyklių arba metodų rinkinį, nurodantį, kaip grupuoti klases į paketus, užtikrintant nuoseklų stilių.

Šio darbo tikslas - identifikuoti ir ivertinti praktikoje naudojamus šablonus kodo skirstymui į paketus. Tai atliekama remiantis moksliniais straipsniais apie sistemos kokybę bei palaikomumą aprašant kriterijus, kurie būtų naudojami šablonams įvertinti, nustatant jų įtaką sistemos palaikomumui.

Tikslui pasiekti yra iškeliami šie uždaviniai:

- Išskirti gerai įgyvendinto kodo požymius
- Aprašyti skirstymo į paketus šablonus, remiantis praktikoje sutinkamais pavyzdžiais
- Pasiūlyti kriterijus, įvertinančius kodo suskirstymo šablono įtaką sistemos kokybei, remiantis rastais gerai įgyvendintos sistemos požymiais
- Pasirinkti kelias sistemas ir pertvarkyti jų failų struktūrą pagal aprašytus šablonus, įvertinant, kiek sudėtinga pasiekti kiekvieno šablono strukūrą
- Naudojant pertvarkytas sistemas, įvertinti kiekvieną kodo skirstymo šabloną pagal pasiūlytus kriterijus
- Pateikti rekomendacijas, kokius šablonus kodo skirstymui tinkamiausia naudoti

Šio darbo metu nagrinėjami ir aprašomi gerai įgyvendinto kodo požymiai, užtikrinantys sistemos stabilumą ir palaikomumą, remiantis Martin Kleppmann *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*, ir Robert C. Martin *Agile Software Development, Principles, Patterns, and Practices* knygomis. Ieškomi kriterijai, kuriuos naudojant galima įvertinti kodo suskirstymo įtaką sistemos kodo kokybei, pavyzdžiui - komponentų skaičius, tiesioginės ir netiesioginės priklausomybės, paketų stabilumas [Mar02]. Tyrinėjami šablonai kodo skirstymui į paketus įvardinti Martin Sadin straipsnyje *Four Strategies for Organizing Code*. Nagri-

nėjamos atviro kodo sistemos, pasirenkant skirtingo tipo projektus, siekiant objektyvesnės šablonų analizės skirtingose srityse. Galimi tipai:

- Taikomoji programinė įranga, teikianti paslaugas įrangos naudotojams. Pavyzdžiui, internetinė programėlė priminimams ir darbams užsirašyti
- Techninė programinė įranga, naudojama taikomosios programinės įrangos duomenų saugojimui, siuntimui, paieškai. Pavyzdžiui, duomenų bazės, pranešimų eilės, talpyklos (angl. cache)
- Programinės įrangos įrankiai, skirti naudoti kitose sistemose supaprastinant programinį kodą, naudojant jau įgyvendintas funkcijas. Pavyzdžiui, Java programavimo kalbos Spring karkasas internetinių programėlių kūrimui

Tyrinėjamų projektų paketų sturktūros pertvarkomos pagal pasirinktus skirstymo šablonus, pertvarkyti projektai įvertinti, naudojant išskirtus kriterijus, nustatant, kokią įtaką skirtingi skirstymo šablonai turi sistemos kokybei.

Likusi šio dokumento dalis yra išdėstyta taip - pirmas skyrius nagrinėja tvarkingos kompiuterinės sistemos sąvoką, kas ją sudaro, kaip galima ją įvertinti, įgyvendinimo kokybę. Aprašyti kriterijai, kaip įvertinti paketų struktūros įtaką sistemos kokybei. Antras skyrius tyrinėja skirtingus šablonus klasėms į paketus skirstyti, jų privalumus bei trūkumus. Trečiame skyriuje aprašomi sukurti įrankiai, reikalingi sistemų analizei ir šablonų įvertinimui, minima, kaip jie įgyvendinti ir kaip jie yra naudojami. Ketvirtame skyriuje analizuojamos pasirinktos atviro kodo sistemos - bandoma nustatyti jų naudojamus šablonus, vertinama sistemų kokybė. Penktame skyriuje aprašomas procesas, kaip pasirinktos sistemos yra perdaromos, kad tiksliai laikytųsi antrame skyriuje aprašytų kodo skirstymų šablonų, įvertinama, kiek sudėtinga pasiekti kiekvieno šablono struktūrą. Nagrinėjama perdarytų sistemų kokybė pagal pirmame skyriuje aprašytus kriterijus, ieškomas geriausiai įvertintas šablonas.

1. Kompiuterinės sistemos vertinimas

1.1. Kompiuterinės sistemos kokybė

Norint išsiaiškinti, kokią įtaką sistemos kokybei daro skirtingi paketų skirstymo metodai ir kaip objektyviai pamatuoti jų įtaką, pirmiausia reikėtų apsibrėžti, kokiais požymiais pasižymi gerai įgyvendinta kompiuterinė sistema. Martin Kleppmann savo knygoje *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems* [Kle17] išskiria šiuos pagrindinius kriterijus:

- Patikimumas, reiškiantis, kad net ir klaidų (įrangos, programinių ar žmogiškųjų) atveju, sistema veikia stabiliai ir patikimai, paslepiant tam tikras klaidas nuo vartotojo [Kle17].
- Prižiūrimumas, reiškiantis, jog skirtingų abstrakcijų pagalba sumažintas sistemos kompleksiškumas. Dėl to nesunku keisti esamą sistemos funkcionalumą bei pritaikyti naujiems verslo naudojimo atvejams. Tai supaprastina darbą inžinierių ir operacijų komandoms, dirbančioms su šia sistema, taip pat leidžia prie sistemos prisidėti naujiems žmonėms, o ne tik jos ekspertams. Tai ypač aktualu atviro kodo sistemoms [Kle17].
- Plečiamumas, reiškiantis, jog sistema turi strategijas, kaip išlaikyti gerą našumą užklausų srautui didėjant ir sistemai augant, tai atliekant su pagrįstais kompiuteriniais resursais ir priežiūros kaina [Kle17].

Yra daug skirtingų elementų, sudarančių sistemą, kuri tenkintų aukščiau paminėtus kriterijus, pavyzdžiui, pasirinktos technologijos, aukšto lygio architektūra, dokumentacija, sistemos testavimo procesai, jų kiekis ir pan. Vienas iš svarbių elementų, prisidedančių prie gerai įgyvendintos sistemos dizaino yra programinio kodo dizainas, jo skaitomumas, patikimumas. Konvencijos, kaip vadinti kodo paketus, kokias klases jiems priskirti ir kokios paketų hierarchijos laikytis sudaro svarbią programinio kodo dizaino dalį. Todėl kuriant programinę įrangą, laikas, skirtas rasti sistemai tinkamą paketų skirstymo šabloną ir to šablono laikytis, atsiperka padarant programinį kodą geriau suprantamu, taip prisidedant prie bendro sistemos dizaino patikimumo ir lengvesnio palaikomumo.

Straipsnyje *Investigating The Effect of Software Packaging on Modular Structure Stability* [Sho19], autoriai akcentuoja, kad gerai įgyvendintos, objektinio stiliaus sistemos turėtų vystytis be didelių pakeitimų jų architektūroje. To siekiama, nes architektūriniai pakeitimai paveikia didelę sistemos dalį ir jų įgyvendinimo ir sistemos priežiūros kaštai yra žymiai didesni[Sho19]. Paketų struktūra, kuri užtikrina atsietą (angl. *decoupled*) komunikavimą tarp paketų TODO performuluoti, enkapsuliuoja paketų vidinius elementus, neleidžiant pakeitimais išplisti už paketų ribų yra pagrindas tvirtai sistemos architektūrai, gebančiai efektyviai plėstis, ženkliai nesikeičiant ir sutaupant programos priežiūros kaštus.

1.2. Kodo skirstymo į paketus metodų vertinimas

Ankstesniame skyriuje buvo nagrinėjama gerai įgyvendintos paketų struktūros įtaka geram kompiuterinės sistemos dizainui, tačiau lieka neatsakytas klausimas - kaip įvertinti metodą kodui į paketus grupuoti, kaip objektyviai užtikrinti, jog pasirinktas sprendimas yra būtent toks, kokio rei-

kia, ir kokia jo įtaka kompiuterinei sistemai? Tvarkingas, aiškiai suprantamas kodas yra subjektyvi tema, priklausanti nuo komandos, naudojamos programavimo kalbos ar programinių įrankių bei programinės sistemos dalykinės srities. Kodo grupavimo į paketus metodai, taip, kaip ir bendros tvarkingo kodo praktikos, gali būti labai subjektyvūs ir patogūs tik metodą formavusiam asmeniui. Tam, kad būtų galima pagrįstai įvertinti skirtingus kodo skirstymo šablonus, pasiekiant kuo objektyvesnį, įtaką sistemos kokybei nusakantį rezultatą, reikėtų aprašyti kriterijus, nusakančius, ko tikimasi iš paketų struktūros.

Robert C. Martin savo knygoje *Agile Software Development, Principles, Patterns, and Practices* [Mar02] aprašo principus, padedančius teisingai grupuoti klases į paketus. Rodiklis, kiek kiekvienas paketas sistemoje laikosi nurodytų principų, tam tikrame paketų skirstymo šablone, gali būti kriterijus vertinant to šablono kokybę ir įtaką bendram sistemos dizainui.

1.2.1. Bendro sąryšio principas

Klasės pakete turėtų būti susietos kartu, kad turėtų tą pačią priežastį pasikeisti. Pakeitimas, kuris paveikia paketą, paveikia visas to paketo klases ir jokių kitų paketų.

Kaip teigia vienos atsakomybės principas (angl. *Single responsibility principle*), klasė turėtų neturėti skirtingų priežasčių keistis, šis principas taip pat teigia, kad paketas taip pat neturėtų turėti skirtingų priežasčių pasikeisti. Principas ragina suburti visas klases, kurios gali keistis dėl tų pačių priežasčių, į vieną vietą. Jei dvi klasės yra taip stipriai susietos, kad jos visada keičiasi kartu, tada jos turėtų būti tame pačiame pakete. Kai reikia išleisti pakeitimus, geriau, kad visi pakeitimai būtų viename pakete. Tai sumažina darbo krūvį, susijusį su pakeitimu išleidimu, pakartotiniu patvirtinimu ir programinės įrangos perskirstymu, be reikalo neatliekant validacijos ir neleidžiant kitų, nesusijusių modulių [Mar02].

1.2.2. Aciklinių priklausomybių principas

Paketo priklausomybės diagramoje neturi būti žiedinių ciklų.

Priklausomybių ciklai sukuria neatidėliotinų problemų. Žiedinės priklausomybės gali sukelti domino efektą, kai nedidelis, lokalus vieno modulio pokytis išplinta į kitus modulius, dėl to, norit testuoti vieną nedidelį modulį, reikia iš naujo sukompiliuoti didžiulę sistemos dalį. Taip pat žiedinės priklausomybės lemia programos ir kompiliavimo klaidas, kadangi pasidaro labai sunku sudaryti tvarką, kaip kompiliuoti paketus. Žiedinės priklausomybės taip pat gali sukelti begalinę rekursiją, kuri sukelia nemalonių problemų tokioms kalboms kaip Java, kurios skaito savo deklaracijas iš sukompiliuotų dvejetainių failų [Mar02].

1.2.3. Stabilių priklausomybių principas

Paketų priklausomybės turetų laikytis stabilumo krypties. Sistema negali būti visiškai statiška. Norint jai plėstis būtinas tam tikras nepastovumas. Kai kurie paketai yra sukurti taip, kad būtų nepastovūs, iš jų tikimasi pokyčių. Nuo paketo, kuris, manoma, yra nepastovus, neturėtų priklausyti sunkiai pakeičiami paketai, nes priešingu atveju nepastovų paketą taip pat bus sunku

pakeisti. Gali susiklostyti situacija, kad modulis, sukurtas taip, kad jį būtų lengva pakeisti, kartais tampa sunkiai keičiamu, pridėjus priklausomybę nuo jo [Mar02].

1.2.4. Stabilių abstrakcijų principas

Paketai turi būti tiek abstraktūs, kiek ir stabilūs. Šis principas nustato ryšį tarp stabilumo ir abstraktumo. Stabilūs paketai turėtų būti abstraktūs, todėl ir lengvai praplečiami. Tai pat šis principas teigia, kad nestabilus paketas turi būti konkretus, nes jo nestabilumas leidžia lengvai pakeisti jo turinio kodą. Taigi, jei paketas yra stabilus, jį taip pat turėtų sudaryti abstrakčios klasės, užtikrinant jo išplečiamumą. Stabilūs paketai, kurie yra lengvai išplečiami, yra lankstūs pakeitimams, nedarant didelės įtakos sitemos struktūrai [Mar02].

1.3. Paketų kokybės matai

Beveik visi autoriaus aprašyti principai turi aiškiai apibrėžtus matus, kuriais galima pamatuoti, kaip stipriai paketas laikosi šių principų. Būtent šie bus naudojamos įvertinti paketus analizuojamuose šablonuose norint patikrinti šablonų kokybę:

- **Klasių skaičius** (N) - klasių skaičiaus matas paketui nurodo, kiek klasių (konkrečių ir abstrakčių) yra pakete. Šis matas matuoja paketo dydį.
- **Aferentinės jungtys (angl. *Afferent Couplings*)** (Af) - aferentinių jungčių matas nurodo skaičių paketų, kurie priklauso nuo klasių, esančių pasirinktame pakete. Šis matas matuoja ateinančias priklausomybes.
- **Eferentinės jungtys (angl. *Efferent Couplings*)** (Ef) - eferentinių jungčių matas nurodo skaičių kitų paketų, nuo kurių priklauso klasės pasirintame pakete. Šis matas matuoja išeinančias priklausomybes.
- **Stabilumas** (S - stabilumo matas nurodo santykį tarp eferentinių jungčių ir visų jungčių (aferentinės + eferentinės) pakete. Šis matas matuoja paketo atsparumą pokyčiams, kuris buvo akcentuojamas stabilių priklausomybių principe:

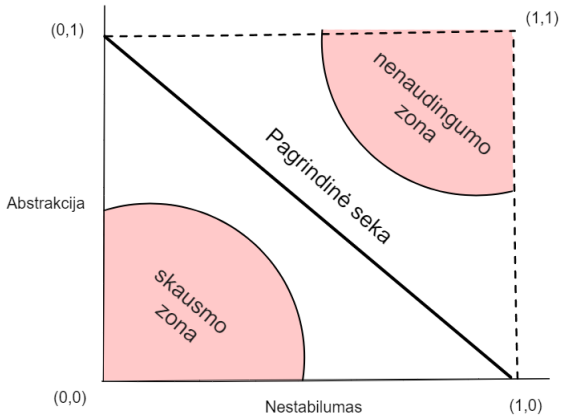
$$Stabilumas(S) = \frac{Jungtys_{eferentines}}{Jungtys_{eferentines} + Jungtys_{aferentines}} \tag{1}$$

  Reikšmės rėžiai - nuo nulio iki vieno, kur vienas nurodo visiškai stabilų paketą, o vienetas - visiškai nestabilų.
- **Abstrakcija** (A) - paketo abstrakcijos matas nurodo santykį tarp abstrakčių klasių (arba sąsajų (angl. *interface*)) pakete ir bendro klasių skaičiaus:

$$Abstrakcija = \frac{N_{abstrakcios}}{N_{visos}} \tag{2}$$

  Šis matas nurodo paketo abstraktumą. Abstrakcijos reikšmė gali būti tarp nulio ir vieno. Nulis reiškia, kad paketas neturi jokių abstrakčių klasių, o vienetas nurodo, kad pakete yra tik abstrakčios klasės.

- **Atstumas nuo pagrindinės sekos** (D) - Pagrindinė seka yra sąryšis tarp nestabilumo ir abstrakcijos. Pagrindinė seka - idealizuota linija (Abstrakcija + Nestabilumas = 1) kurią galima vaizduoti kaip kreivę, su abstrakcijos dydžiu y ašyje ir nestabilumu x ašyje. Paketas tiesiai ant pagrindinės sekos yra optimaliai subalansuotas atsižvelgiant į jo abstraktumą ir stabilumą. Idealūs paketai yra arba visiškai abstraktūs ir stabilūs (Stabilumas=0, Abstrakcija=1) arba visiškai konkretūs ir nestabilūs (Nestabilumas=1, Abstrakcija=0).



1 pav. Pagrindinės sekos kreivė

Atstumą nuo pagrindinės sekos galima apskaičiuoti kaip:

$$Atstumas = |Abstracija + Nestabilumas - 1| \tag{3}$$

Šis matas yra paketo abstraktumo ir stabilumo pusiausvyros rodiklis. Šio mato diapazonas yra nuo nulio iki vieno, kur nulis reiškia paketą, sutampantį su pagrindine seka, o vienas - paketą, maksimaliai nutolusį nuo pagrindinės sekos.

- **Žiedinės priklausomybės** (C) - žiedinių priklausomybių matas skaičiuoja atvejus, kur pasirinkto paketo išeinančios priklausomybės taip pat yra paketo ateinančios priklausomybės (tiesiogiai arba netiesiogiai). Šis matas - aciklinių priklausomybių rodiklis, minėtas aciklinių priklausomybių principe.

Šių matų patikimumas buvo ivertintas atvejo analizėje *Exploring the Relationships between Design Metrics and Package Understandability: A Case Study*, kurioje autoriai tyrinėjo sąryšį tarp minėtų matų ir vidutinių pastangų, reikalingų suprasti objektinio dizaino paketą. Tyrimas atliktas naudojant aštuoniolika paketų, paimtų iš dviejų atviro kodo programinės įrangos sistemų. Paskaičiuoti pastangas, reikalingas paketui suprasti, buvo pasitelktos trys skirtingos komandos, turinčios po tris, panašią patirtį turinčius porgramuotojus. Jų buvo paprašyta pilnai suprasti paketų funkcionalumą ir nuo vieno iki dešimt įvertinti pastangas, reikalingas suprasti kiekvieną paketą. Rezultatai gauti iš šio tyrimo rodo statistiškai reikšmingą koreliaciją tarp daugumos matų ir paketų suprantamumo [Eli10].

2. Galimi kodo skirstymo į paketus šablonai

Diskusijose, kaip reikėtų skirstyti programinį kodą, paprastai akcentuojami du metodai – pagal *techninį sluoksnį*, kur kiekvienam funkcionalumui arba kompiuterinės sistemos sluoksniui yra sukuriamas paketas, grupuojant skirtingų dalykinių sričių esybes, arba pagal *dalykinės srities esybes*, kur vienos esybės kodas, dalykinės srities esybės funkcionalumas skirtingose programiniuose sluoksniuose yra patalpintas viename pakete [Jan21] Tačiau šie du metodai yra gan platūs ir, dažniausiai jų nėra griežtai laikomasi – klasės būna išskaidytos remiantis papildomomis taisyklėmis, siekiant išspresti sistemos planavimo metu kylančias problemas. Norint išskirti šias papildomas taisykles, šablonus, buvo nagrinėjamos atviro kodo sistemos, stebimi nukrypimai nuo bendresnio kodo skirstymo būdo ir identifikuojami klausimai arba problemos, kurias buvo bandoma išspresti.

2.1. Sistemų analizės procesas

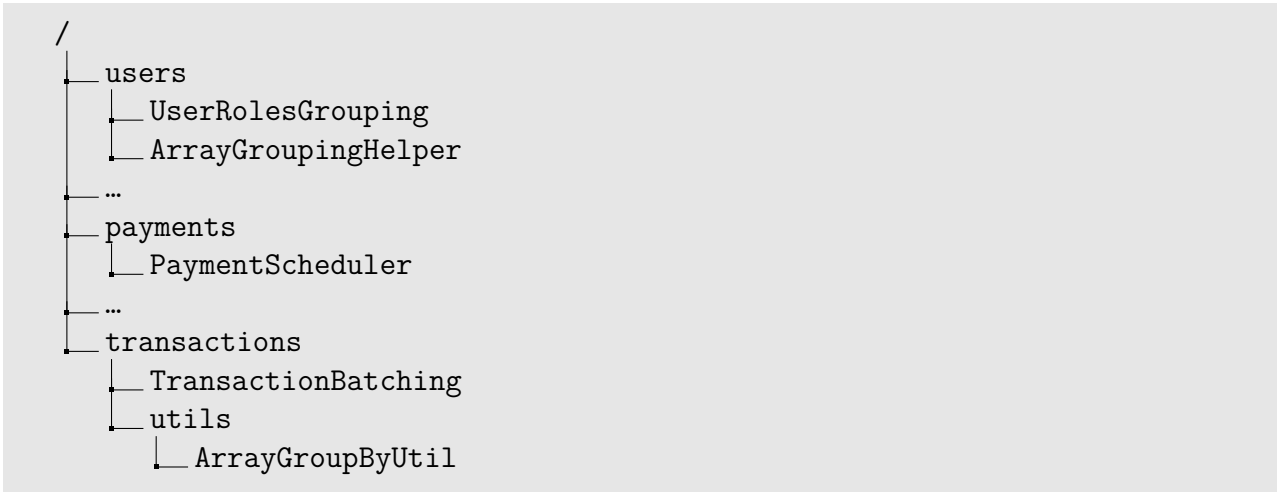
todo: išsiplėsti apie sistemų analizės procesą

2.2. Problemos

Iš analizuotų sistemų buvo rastos šios problemos, būdingos sistemų dizainui, kartu su jas sprendžiančiais šablonais ir pavyzdžiais, kokiose sistemose jie yra:

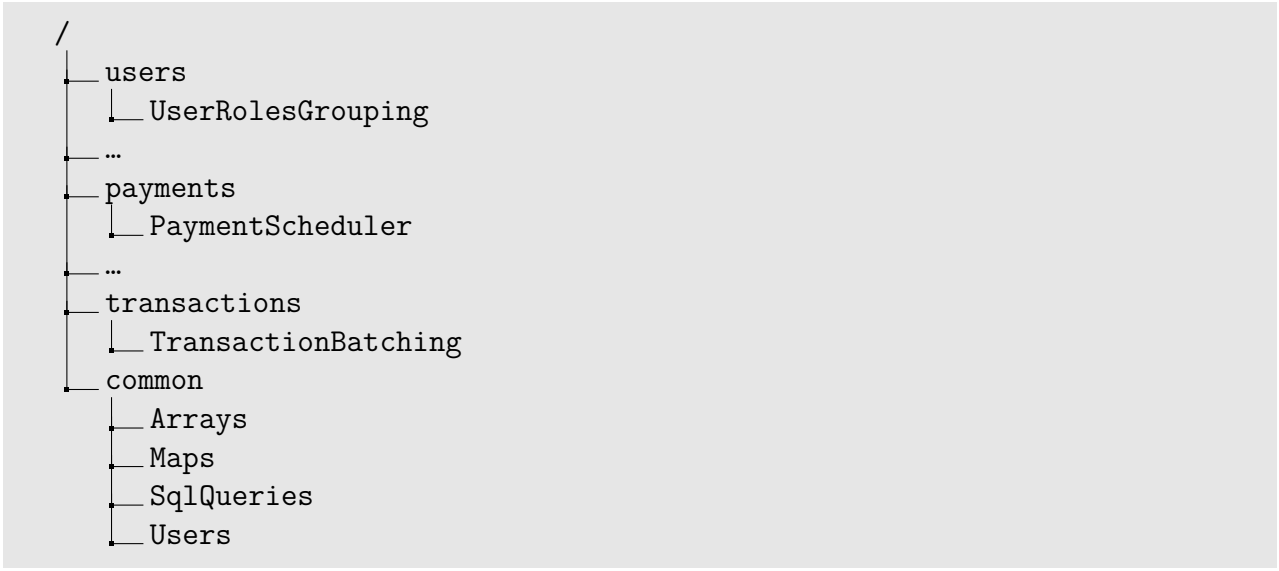
2.2.1. Pagalbinių, daugkartinio naudojimo klasių skirstymas

Pagalbinės ir daugkartinio naudojimo klasės dažniausiai negali būti patogiai grupuojamos skirstant pagal domeno esybes – pagalbines klases gali naudoti kelios esybės, todėl neaišku, prie kurios jas reikėtų priskirti. Skirstant pagal techninį sluoksnį, bazines pagalbines klases gali naudoti keli sluoksniai. Didžiausia problema, susijusi su pagalbinėmis klasėmis, kurios turėtų išspręsti dažnai sistemoje sutinkamas problemas – inžinieriai, dirbantys prie sistemos nežino apie jų egzistavimą, todėl jų nenaudoja, tai veda prie didesnio kodo pasikartojimo arba kelių skirtingų to paties pagalbinio funkcionalumo įgyvendinimo.



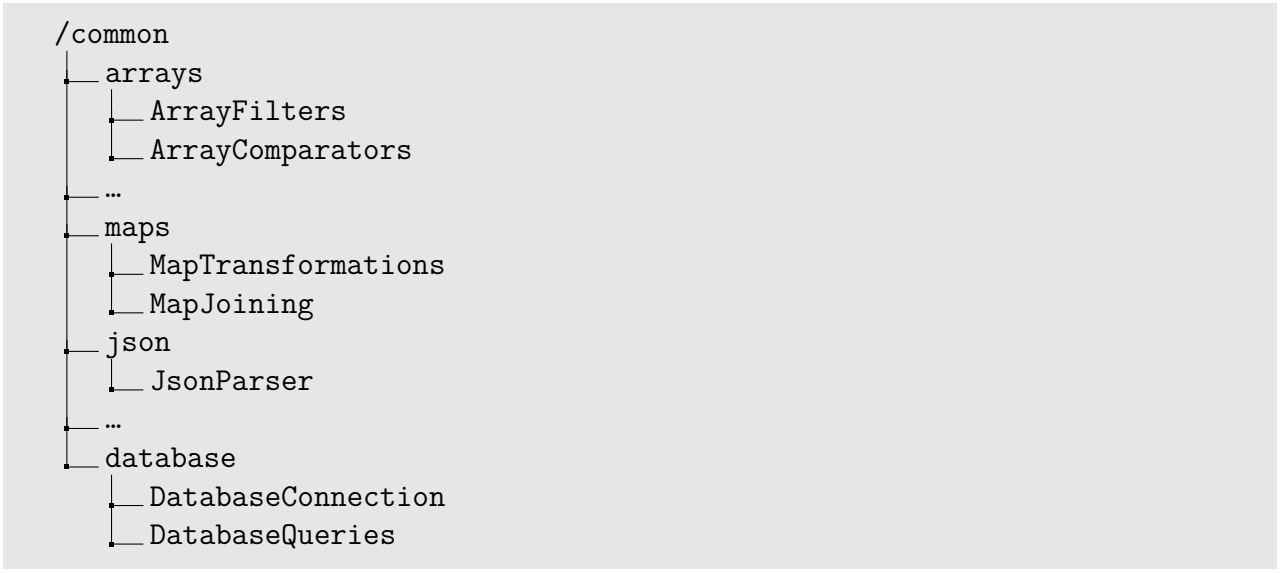
2 pav. Sistemos pavyzdys, kur labai panašią funkciją atliekančios klasės *ArrayGroupByUtil* ir *ArrayGroupingHelper* egzistuoja todėl, kad inžinierius nerado jau įgyvendintos klasės, dėl aiškios struktūros daugartinio panaudojimo klasėms trūkumo.

Vienas iš šablonų, sprendžiančių šią problemą, galėtų būti turėti vieną paketą, skirtą visoms pagalbinėmis klasėmis, kuris yra paminėtas sistemos dokumentacijoje ir apie jo egzistavimą teoriškai žino visi komandos nariai. Pakete reikėtų turėti atskiras klases kiekvienam bendriniam domenui, iš kurios pavadinimo programuotojas galėtų nuspresti, kad jo ieškomas funkcionalumas, bus būtent toje klasėje.



3 pav. Sistemos pavyzdys, kur visas bendrinio panaudojimo kodas guli *common* pakete, pirmame sistemos paketų lygyje, todėl pagalbinės klasės yra lengvai randamos.

Jei pagalbinių klasių dydis labai išauga, jas galima sumažinti ir vietoj vienos atskiros klasės vienai bendrinei sričiai, sukurti vieną paketą, ir jame turėti kelias pagalbines klases, susijusias su tuo domenu. Tokiu atveju reikia užtikrinti, kad iš klasių pavadinimo aišku, kokį srities subdomeną padengia klasė.



4 pav. Sistemos pavyzdys, kur bendrinio panaudojimo kodas guli *common* pakete, po domeno subpaketais, taip sumažinant klasių dydį

Tokį pagalbinių klasių skirstymo metodą naudoja keletas repozitorijų - pavyzdžiui, užrašinės aplikacijos *Omni-Notes*[1] bei *Fire Sticker*[2]

Naudojant tokį šabloną, programuotojas, susiduriantis su bendrine problema, kuri, labai tikėtina, jau yra išspręsta sistemoje turėtų aiškų procesą, kaip elgtis šioje situacijoje:

1. Atsidaryti vieną paketą, skirtą bendrinio panaudojimo kodui
2. Pakete surasti klasę, kurios pavadinimas būtų susijęs su jo problema
3. Klasės funkcijų saraše surasti jam tinkamą funkciją.
4. Jei reikalingas funkcionalumas nerastas, įgyvendinti jį pasirinktoje klasėje, padengti jį testais, bei aprašyti dokumentaciją, kaip funkcija turėtų būti naudojama.
5. Iškviesti rastą arba sukurtą funkciją iš bendrinio panaudojimo kodo paketo savo funkcionalume

2.2.2. Ka daryti esant dideliam priklausomybių nuo paketo skaičiui?

Didelis priklausomybių nuo specifinio paketo skaičius (arba aferentinės jungtys), reiškia, kad pokyčiai tame pakete turės įtaką kelioms klasėms. Jei tokia tendencija yra būdinga visai sistemai, sistema tampa mažiau lanksti pokyčiams, kadangi net ir paprastas pakeitimas daro įtaką reišmingai sistemos daliai, pokyčiai yra labiau linkę keisti bendrą sistemos architektūrą. Taip pat naujos sistemos versijos išleidimo (angl. *release*) procesas tampa sudetingesnis, kadangi yra paveikiama daugiau klasių.

Robert C. Martin bendro sąryšio principas, kuris teigia, kad visos tarpusavyje susijusios klasės turėtų būti vienam pakete, akcentuoja siekiamybę turėti gan mažus paketus, turinčius

[1] `https://github.com/federicoiosue/Omni-Notes/tree/develop/omniNotes/src/main/java/it/feio/android/omninotes/utils`

[2] `https://github.com/hackjutsu/Fire_Sticker/tree/master/app/src/main/java/com/gogocosmo/cosmoqiu/fire_sticker/Utils`

aiškiai apibrėžtą funkcionalumą, priežastį egzistuoti, taip užtikrinant glaudų tarpusavyje susijusių klasių saryšį. Šis principas galėtų būti kodo skirstymo šablonas, užtikrinantis racionalų aferentinių jungčių skaičių paketuose.

Vadovaujantis šiuo šablonu kiekvieną paketą reikėtų realizuoti kaip komponentą, teikiantį vieną funkcionalumą. Funkcionalumas kitiems paketams yra pasiekiamas per vieną minimalią sąsają (angl. *interface*), kuri atskleidžia tik konceptus (metodus arba duomenų tipus), kurie yra glaudžiai susiję su komponento teikiama paslauga, bei klase, gražinančią minėtos sąsajos įgyvendinimą. Tai gali būti paprasta klasė, su statine funkcija, kurios rezultatas yra ši sąsaja, arba, esant keliomis sąsajos implementacijomis, *Static factory* dizaino šablonas, kuris nusprendžia kuri įgyvendinimą reiketų grąžinti pagal paduotus argumentus.

Visos kitos klasės naudoja *package* pasiekiamumo modifikatorių, taip kompiliatoriaus pagalba užtikrinant, kad jos nebus pasiektos iš išorės.

Paketas turintis vieną funkciją yra naudojamas tik tų paketų, kuriems reikia būtent tos funkcijos, taip užtikrinant tik mažos sistemos dalies priklausomybę nuo vieno paketo.

Toks konceptas yra sutinkamas *Typescript* programavimo kalboje, kur kiekvienas modulis (šios kalbos paketo atitikmuo) turi *index.ts* faila, veikiantį kaip sąsaja, apibrėžiančia, kokios modulio klasės bei funkcijos gali būti pasiekiamos už paketo ribų. Tai užtikrina glaustą kontraktą ir aiškų modulio funkcionalumą, kurį gali pasiekti kiti moduliai, bei paslepia modulio įgyvendinimo detales. Toki šabloną naudoja turinio valdymo sistema *Keystone*[3], kur kiekvieną esybė turi savo modulį, atliekantį vieną funkcija, kurio kontraktas aprašytas *index.ts* faile.

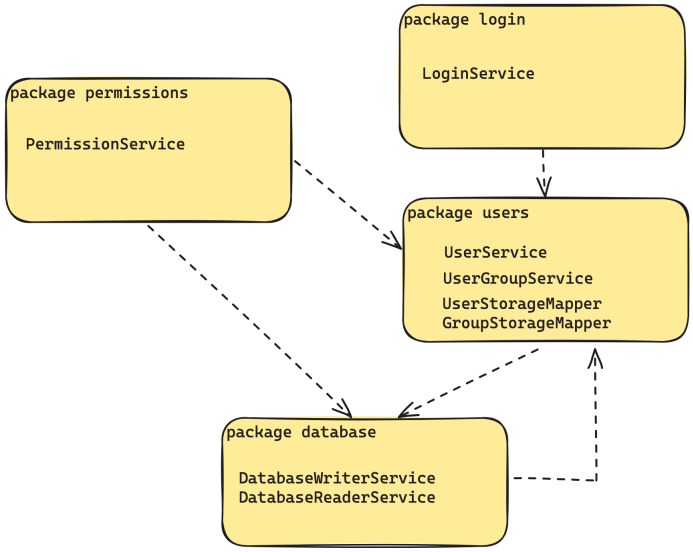
```
/types
├── image
│   ├── views
│   ├── utils
│   └── index
├── text
│   ├── views
│   └── index
├── ...
└── checkbox
    ├── views
    └── index
```

5 pav. *Keystone* sistemos dalis, kurioje matomi smulkūs moduliai su *index.ts* failuose aprašytais kontraktais

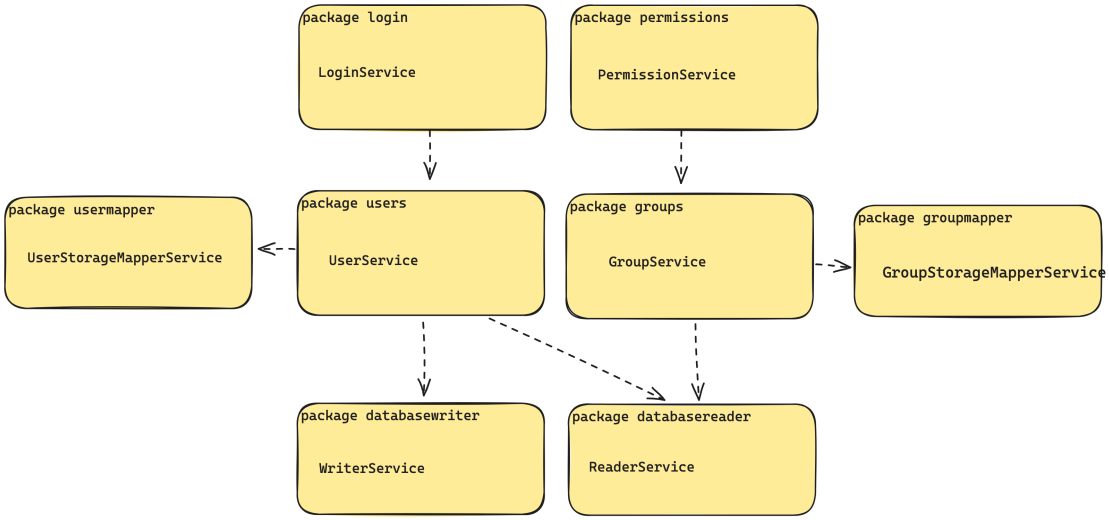
Taip pat mažas paketo funkcionalumas reiškia, kad minėtas paketas skirtas funkcionalumui įgyvendinti naudos minimalų kitų sistemos esybių skaičių, taip sumažinant ir eferentinių jungčių skaičių.

Žemiau esančiuose paveikslėliuose galima matyti, kaip išskaidant paketus, turinčius kelis funckionalumus, yra sumažinamas paketų priklausomybių skaičius.

[3] `https://github.com/keystonejs/keystone/tree/main/packages/core/src/fields/types`



6 pav. Sistemos pavyzdys su kelias funkcijas atliekančiais paketais



7 pav. Sistemos pavyzdys su aiškią, vieną funkciją turinčiais paketais

Toks skirstymo būdas taip pat sprendžia ciklinių priklausomybių problemą - pavyzdžiui, įrankio, skirto spręsti ciklinių priklausomybių problemą, kūrimo aprašas [Aga15] teigia, kad ciklinių priklausomybių problemą galima spręsti laikantis trijų principų - bendro panaudojimo, bendro keitimosi bei paleidimo ir pernaudojimo ekvivalentumo. Šie principai yra išvesti iš bendro saryšio principo ir akcentuoja, kad kartu besikeičiančios klasės turėtų būti viename pakete. Norint užtikrinti šiuos reikalavimus, reikėtų turėti mažą, tiksliai apibrėžto funkcionalumo paketą, kurio visi komponentai - tarpusavyje susiję. Tokiu atveju ciklinių priklausomybių tikimybė sumažėja.

2.2.3. Ka daryti su daug skirtingų sąsajų implementacijų?

Sistemai plečiantis, galima susidurti su problema, kad išauga sąsajos implementacijų skaičius. Ši problema gali iškilti tiek bandant skirstyti kodą pagal domeno esybes, tiek pagal techninį sluoksnį. Skirstant pagal domeną, gali būti neaišku, kur turėtų būti sukuriamas naujas esybės paketa. Skirstant kodą pagal techninį sluoksnį, gali susidaryti klasių perteklius, pavyzdžiui, model

pakete. Tokiu atveju navigacija pakete pasidaro sudėtinga, neaišku, kas implementuoja. Šią problemą būtų galima spręsti sukuriant sąsajos implementacijas tame pačiame pakete kaip ir pati sąsają. Toks skirstymo į paketus šablonas yra sutinkamas *Java* standartinėje bibliotekoje[4], kur visos standartinės sąsajos bei jų implementacijos yra tame pačiame pakete. Dėl tokios paketų struktūros, inžinieriai, naudojantys standartinę *Java* biblioteką, gali lengvai rasti jiems reikiamus sąsajos įgyvendinimus:

```
/util
├── Map
├── HashMap
├── LinkedHashMap
├── WeakHashMap
├── TreeMap
├── ...
├── List
├── LinkedList
└── ArrayList
```

8 pav. *Java* standartinės bibliotekos pavyzdys, kuriame sąsajų įgyvendinimai guli tame pačiame pakete, kaip ir sąsajos

Šis šablonas turi vieną trūkumą - pakete su didelių klasių kiekių, bei keliom skirtingom sąsajom, sunku suprasti, kuri klasė, kurią sąsaja įgyvendina. Tai pastebima ir *Java* bibliotekos pavyzdyje - *utils* paketas išviso turi viršs 50 klasių, iš kurių bent 10 yra sąsajos, o visa kita jų implementacijos, todėl greitai rasti norima sąsajos įgyvendinima vistiek užtrunka.

Šio šablono trūkumas gali būti išspręstas, ji šiek tiek pakeitus - sąsaja, kuri turi daug skirtingų įgyvendinimų turėtų turėti specifiškai jai sukurtą paketą, o kiekvienam paketo įgyvendinimui sukuriamas po subpaketas, kuriame yra tiek klasė įgyvendinanti minėta sąsają, tiek kitos įgyvendinimui reikalingos klasės. Naudojant tokia kodo struktūra galima labai greitai rasti sąsasas, bei galimus jos įgyvendinimus, beveik netyrinėjant sistemos struktūros, bei klasių kodo. Šis šablonas taip pat užtikrina, kad paketas turį vieną aiškų funkcionalumą todėl laikosi bendro sąryšio principo. Šablone aprašyta paketų struktūra buvo sutikta tokiose sistemose kaip *HikariCP*[5], skirtoje efektyviam *Java* programos prisijungimui prie duomenų bazių, bei *Leaf*[6], naudojamoj unikalaus identifikatoriaus generavimui.

[4] https://github.com/AdoptOpenJDK/openjdk-jdk11/tree/master/src/java.base/share/classes/java/util

[5] https://github.com/brettwooldridge/HikariCP/tree/dev/src/main/java/com/zaxxer/hikari

[6] https://github.com/Meituan-Dianping/Leaf/tree/master/leaf-core/src/main/java/com/sankuai/inf/leaf

```
/
├── segment
│   ├── dao
│   ├── model
│   └── SegmentIdGenImpl
├── snowflake
│   ├── SnowflakeIdGenImpl
│   └── SnowflakeZookeeperHolder
└── IdGen
```

9 pav. *Leaf* sistema, kur kiekvienas sąsajos įgyvendinimas yra aprašytas subpakete

2.2.4. Kaip valdyti esybių pokyčius ir versijavimą

Sistemai egzistuojant ilgesnį laiką, jos pokyčiai pasidaro neišvengiami. Smulkūs pokyčiai nebūtinai paveikia bendrą sistemos struktūrą, tačiau didesniems pokyčiams kartais reikia sukurti naujas klasių ar funkcionalumų versijas. Jei reikia palaikyti atgalinį suderinamumą (angl. backward compatibility), sistemoje gali atsirasti kelios tų pačių esybių versijos. Tokiu atveju kelios tų pačių esybių versijos sistemoje gali pridėti painumo. Ši problema dažnai sprendžiama sukuriant atskirus paketus skirtingoms versijoms (v1, v2, v3, ...) bei iškeliant bendrą, abiejų esybių naudojama kodą į subpaketus taip, kad jas galėtų pasiekti abi versijos. Taip galima išvengti kodo duplikacijos bei perteklinio klasių skaičiaus paketuose, bei užtikrinant aiškia skirtingų versijų atskirti.

```
/regex
├── common
├── pending
├── v3
├── v4
└── ...
```

10 pav. *Mongo* duombazės Sistemos pavyzdys, kurioje skirtingos versijos patalpintos atskiruose paketuose

Toks skirstymo būdas matomas keliose repozitorijose - *Mongo*[7] duomenų bazėje, API skirtame kelionių valdymui *travels-java-api*[8] bei duomenų saugojimo įrankyje *nocodb*[9]. Šiose repozitorijose v1, v2 pavadinti paketai saugo skirtingas esybių versijas.

Todo: parašyti, kad galima išrinkti dar daugiau problemu bei šablonu tesiant analize Todo: paaiškinti, kad naudojant matus šablonams įvertinti, atmetam dali šablonų, kurie negali būti jais pamatuoti Todo; pridėti šabloną sprendžianti distance

[7] `https://github.com/mongodb/mongo/tree/master/src/third_party/boost/boost/regex`

[8] `https://github.com/mariazevedo88/travels-java-api/tree/master/src/main/java/io/github/mariazevedo88/travelsjavaapi/controller`

[9] `https://github.com/nocodb/nocodb/tree/c7cc1f92fd77f8b5daefceb7148aab4a69cb9b4e/packages/nocodb/src/meta/migrations`

3. Įrankiai šablonų analizei ir įvertinimui

Kompiuterinės sistemos, kurioms yra aktualu klasių ir paketų skirstymo metodai, paprastai yra labai didelės. Pilnai perprasti tokias sistemas, nustatyti jų įgyvendinimo kokybę, naudojamus šablonus kodui skirstyti, apskaičiuoti paketų kokybės matus yra sudėtingas procesas. Daryti tai rankiniu būdu užtrunka daug laiko bei yra paliekama daug vietos potencialioms klaidoms, todėl yra būtina šį procesą optimizuoti, skaitmenizuoti analizės procedūras pasitelkiant aiškiai apibrėžtų ir programiškai efektyvių programinių įrankių pagalbą.

3.1. Reikalavimai įrankiams

Įrankių, leidžiančių paprasčiau atlikti sistemų analizę, atsakomybes galima suskirstyti į dvi grupes:

- Bendrinė sistemos analizė – įrankis ar įrankiai padeda atlikti bendrinę sistemos analizę. Šios atsakomybių grupės įrankių išvestis nėra objektyvūs, tiksliai apibrėžtų formulių rezultatai, o papildoma, aiškiai pavaizduota meta informacija apie sistemą – paketų struktūrą, jų priklausomybes, figuruojančių paketų bei klasių vardus. Ši papildoma informacija nėra aiškūs teiginiai, o tik pagalba analizę atliekančiams asmeniui, leidžianti priimti išvalgas apie sistemą, kaip pavyzdžiui, kokiam paketų skirstymo šablonui yra atimiausia sistemos strukūra, arba kaip lengvai sistema yra suprantama.
- Paketų kokybės matų skaičiavimas – įrankis ar įrankiai turi padėti apskaičiuoti aprašytus paketų kokybės matus. Šių įrankių išvestis – tikslūs, formulėmis pagrįstų skaičiavimų rezultatai apie paketų kokybę, kuriuos galima lyginti tarpusavyje.

3.2. Reikalavimai bendrinės sistemos analizės įrankiui

Įrankis bendrinei sistemos analizei atlikti turėtų suteikti galimybę naudotojui nurodyti kelią iki *Java* programavimo kalba parašytos sistemos arba posistemės ir joje atlikti jos turinio analizę bei naudotojui pateikti naudingas išvadas, sudarytas iš:

- Klasių ir paketų skaičiaus
- Vidutinio klasių pakete skaičiaus
- Paketų ir klasių medį, identifikuojantį abstrakčias klases ar sąsajas
- Paketų priklausomybių grafiką

Gautą rezultatą išvesti suprantamu formatu, leidžiant vartotojui susidaryti išvadas apie sistemos, arba tam tikros posistemės struktūrą, naudojamus įrankius bei kokybę.

3.3. Reikalavimai įrankiui paketo kokybei skaičiuoti

Įrankis paketo kokybei skaičiuoti turėtų suteikti galimybę naudotojui nurodyti kelią iki *Java* programavimo kalba parašytos sistemos arba posistemės ir joje apskaičiuoti kiekvieno paketo kokybės matus:

- Klasių skaičių
- Aferentinių jungčių skaičių
- Eferentinių jungčių skaičių
- Nestabilumo santykį
- Abstrakcijos santykį
- Atstumo nuo pagrindinės sekos santykį
- Žiedinių priklausomybių skaičių

Gautą rezultatą išvesti vartotojui suprantamu formatu, kuriame matytųsi individualių paketų matai, bei šių matų vidurkis sistemoje (arba posistemėje). Išvedimo formatas turėtų būti toks, jog skirtingų analizių rezultatai būtų lengvai palyginami su kitais.

Abiejose vienas iš palaikomų išvesties formatų turėtų būti *latex*, taip suteikiant galimybę analizės rezultatus pateikti tolesniame šio dokumento turinyje.

3.4. Įrankių įgyvendinimas

Nors beveik visiems reikalavimuose minimiems funkcionalumams galima rasti jau sukurti įrankių, greit ir efektyviai pritaikyti juos skirtingoms sistemoms (arba posistemėms) nėra patogu - kiekvieną įrankį reikėtų vykdyti atskirai, su skirtingais vykdymo procesais ir argumentais. Taip išvestų rezultatų formatai yra skirtingi. Todėl, norint palengvinti ši procesą - suvienodinti procesų vykdymą bei gautus rezultus, visi įrankiai reikalingi analizei įgyvendinti kaip viena programinė sistema, kuri apdoroja failus nurodytoje sistemos direktorijoje, nuskaito *java* failų turinį ir sukonstruoja informaciją apie sistemos paketus bei klases. Surinkta informacija naudojama įgyvendinti kiekvienam aprašytam įrankio funkcionalumui, ten, kur galima, naudojant jau parašytus įrankius, taip programiškai supaprastinant skirtingų įrankių vykdymą.

4. Esamų sistemų pertvarkymas

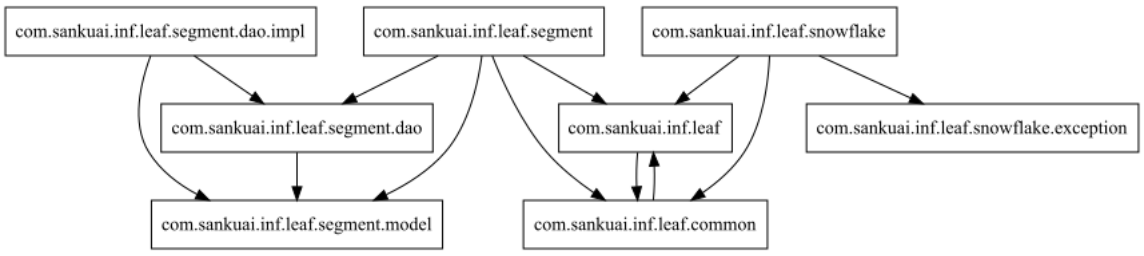
Apibrėžus šablonus paketų skirstymui galima pradėti vertinti jų efektyvumą, juos pritaikant esamoms sistemoms. Šio skyriaus tikslas - pasirinkti ir išnagrinėti kelias sistemas, pasitelkus paketų kokybes matus, bei bendrą sistemos struktūros analizę, taip identifikuojant programiniam kodui būdingas problemas, rastas problemas išspręsti pritaikant aprašytus šablonus. Atlikus pertvarkymus sistemose, dar kartą paskaičiuoti paketų kokybės matus, taip gaunant įrodymus, ar gauti šablonai yra efektyvūs ir iš tiesų sprendžia jiems priskirtas problemas.

4.1. Sistemų pasirinkimas

Sistemų pertvarkymui pasirinktos atviro kodo sistemos, kurių kodas yra viešai prietinamas *github* platformoje. Pasirinktos sistemos yra vidutinio dydžio, todėl nėra labai sudėtinga jas suprasti ir pertvarkyti, bet taip pat jos nėra tokios paprastos, kad neturėtų sistemos dizaino problemų. Pasirinktos sistemos yra skirtingo tipo projektai, taip užtikrinant didesnę problemų ivairovę ir objektyvesnius įvertinimus. Per visą pasirinktų sistemų imtį yra sutinkamos visos aprašytos problemos, taip įvertinant visus aprašytus šablonus.

4.2. *Leaf* sistema

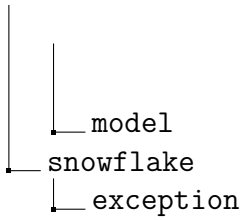
Leaf[10] sistema, per *http* protokolą teikiantį aplikacijų programavimo sąsają unikalaus identifikatoriaus generavimui. Ši sistema užtikrina identifikatoriaus unikalų pasiskirstymą, tarp skirtingų, paskirstytų sistemų (angl. *distributed systems*), servisais orientuotoje (angl. *service-oriented*) architektūroje. Tai techninės programinės įrangos tipas, kuris naudojamas kitų taikomosios programinės įrangos sistemų. Sistema Leaf susideda iš dviejų modulių *server* ir *core*, šiame darbe dėmesys bus skirtas tik *core* moduliui, kadangi *server* modulis yra labai mažas ir paketų struktūra jame neatlieka esminio vaidmens. Prieš visus pakeitimus *Leaf* sistemos, *core* modulio paketų struktūra atrodo taip:



11 pav. *Leaf* sistemos *core* modulio struktūra

```
/
├── common
├── segment
│   └── dao
│       └── impl
```

[10] https://github.com/Meituan-Dianping/Leaf/tree/master



Paketo vardas	N	A	E	S	A	D
leaf.segment.dao.impl	1	0	2	1.0	0.0	0.0
leaf.segment	1	0	4	1.0	0.0	0.0
leaf.snowflake.exception	3	1	0	0.0	0.0	1.0
leaf.segment.model	3	3	0	0.0	0.0	1.0
leaf.segment.dao	2	2	1	0.333	1.0	0.333
leaf	1	3	1	0.25	1.0	0.25
leaf.common	6	3	1	0.25	0.0	0.75
leaf.snowflake	2	0	3	1.0	0.0	0.0

$\bar{N}$	$\bar{A}$	$\bar{E}$	$\bar{S}$	$\bar{A}$	$\bar{D}$
2	2	2	0.479	0.25	0.417

Rezultatai

Rezultatų skyriuje išdėstomi pagrindiniai darbo rezultatai: kažkas išanalizuota, kažkas sukurta, kažkas įdiegta. Tarpinių žingsnių išdavos skirtos užtikrinti galutinio rezultato kokybę neturi būti pateikiami šiame skyriuje. Kalbant informatikos termi- nais, šiame skyriuje pateikiama darbo išvestis, kuri gali būti įvestimi kituose panašios tematikos darbuose. Rezultatai pateikiami sunumeruotų (gali būti hierarchiniai) sąrašų pavidalu. Darbo rezultatai turi atitikti darbo tikslą.

Išvados

1. Išvadų skyriuje daromi nagrinėtų problemų sprendimo metodų palyginimai, siūlomos rekomendacijos, akcentuojamos naujovės.
2. Išvados pateikiamos sunumeruoto (gali būti hierarchinis) sąrašo pavidalu.
3. Darbo išvados turi atitikti darbo tikslą.

Šaltiniai

[Aga15] B. Aga. A Tool for Breaking Dependency Cycles Between Packages. 2015 [žiūrėta 2024-04-04]. Prieiga per internetą: https://scg.unibe.ch/archive/masters/Aga15a.pdf.

[Eli10] M. Elish. Exploring the Relationships between Design Metrics and Package Understandability: A Case Study. 2010 [žiūrėta 2024-03-04]. Prieiga per internetą: https://www.researchgate.net/publication/221219583_Exploring_the_Relationships_between_Design_Metrics_and_Package_Understandability_A_Case_Study.

[Jan21] M. Jang. Two Different Ways To Package Your Code. 2021 [žiūrėta 2024-04-04]. Prieiga per internetą: https://medium.com/ryanjang-devnotes/two-different-ways-to-package-your-code-e9cb12d1b6ea.

[Kle17] M. Kleppmann. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly Media, 2017.

[Mar02] R. C. Martin. *Agile Software Development, Principles, Patterns, and Practices*. Pearson, 2002.

[Sho19] M. A. Shouki A. Ebad. Investigating The Effect of Software Packaging on Modular Structure Stability. 2019 [žiūrėta 2024-03-04]. Prieiga per internetą: https://d1wqtxts1xzle7.cloudfront.net/75578419/pdf-libre.pdf?1638471828=&response-content-disposition=inline%3B+filename%3DInvestigating_the_Effect_of_Software_Pac.pdf&Expires=1709553794&Signature=BWU2DSSXDNujfvTllUWYspcqNlbW1Fgg~doSlc6JoeK7XXJ5bGLPB1B1yBD0tnojJ0yNuWZzQP9fpTjd~yffOhlxnM4GyB2gNMuGZyXsDBXWQuD66kZpwWdluJ63GWjvs28T2ArRpaekqk9JAc-Icun18nyonYJ~W4pIViibjHA4k9yN5r1FZRSWFeUSHpRmflEpJ1opD2Nh889TquLxsA2DhPP3L5_&Key-Pair-Id=APKAJLOHF5GGSLRBV4ZA.