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. Vardauskas Pavardauskas

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 ..... 6

1. KOMPIUTERINĖS SISTEMOS VERTINIMAS ..... 8
   1.1. Kompiuterinė sistemos kokybė ..... 8
   1.2. Kodo skirstymo paketais metodų vertinimas ..... 8
      1.2.1. Bendro sąryšio principas ..... 9
      1.2.2. Aciklinių priklausomybių principas ..... 9
      1.2.3. Stabilių priklausomybių principas ..... 9
      1.2.4. Stabilių abstrakcijų principas ..... 10
   1.3. Paketų kokybės metrikos ..... 10

2. GALIMI KODO SKIRSTYMO Į PAKETUS ŠABLONAI ..... 12
   2.1. Pagal komponentą ..... 12
   2.2. Problemos ..... 13
      2.2.1. Pagalbinių, daugkartinio naudojimo klasių skirstymas ..... 13
      2.2.2. Ka daryti esant dideliam priklausomybių nuo paketo skaičiui? ..... 15
      2.2.3. Ka daryti su daug skirtingų sąsajų implementacijų? ..... 17
      2.2.4. Ka daryti su mikroservisų architektūra? ..... 17
      2.2.5. Ka daryti su paslėptom priklausomybėm? ..... 17
      2.2.6. Kaip grupuoti kodą mono repozitorijoj? ..... 17
      2.2.7. Ka daryti su greitai besikeičiančiu kodu? ..... 17
      2.2.8. Kaip valdyti esybių pokyčius ir versijavimą ..... 17

3. ĮRANKIAI ŠABLONŲ ANALIZEI IR ĮVERTINIMUI ..... 21
   3.1. Reikalavimai įrankiams ..... 21
   3.2. Reikalavimai bendrinės sistemos analizės įrankiui ..... 21
   3.3. Reikalavimai įrankiui paketo kokybėi skaičiuoti ..... 21
   3.4. Įrankių įgyvendinimas ..... 22

4. KODO SKIRSTYMO METODAI REALIOSE SISTEMOSE ..... 23
   4.1. Sistemų pasirinkimas ..... 23
   4.2. Sistemų analizės procesas ..... 23

5. SISTEMŲ PERTVARKYMAS PAGAL ŠABLONUS ..... 24

6. MEDŽIAGOS DARBO TEMA DĖSTYMO SKYRIAI ..... 25
   6.1. Poskyris ..... 25
   6.2. Faktorialo algoritmas ..... 25
      6.2.1. Punktas ..... 25
         6.2.1.1. Papunktis ..... 25
      6.2.2. Punktas ..... 25

7. SKYRIUS ..... 26
   7.1. Poskyris ..... 26
   7.2. Poskyris ..... 26

REZULTATAI ..... 27

IŠVADOS ..... 28

ŠALTINIAI ..... 29

SANTRUMPOS ..... 30

PRIEDAI ..... 31
1 priedas. Neuroninio tinklo struktūra ..... 31
2 priedas. Eksperimentinio palyginimo rezultatai ..... 32

# Įvadas

Teisingai įgyvendintas kompiuterinės sistemos dizainas yra vienas iš kritinių sėkmingo verslo elementų. Tam, jog 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 šablonus kodo skirstymui į paketus. Remiantis moksliniais straipsniais apie sistemos kokybę bei palaikomumą aprašyti 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 pavyzdžiais teorinėje medžiagoje
- Įvertinti, kiek realių sistemų struktūra nutolusi nuo teorinių šablonų apibrėžimų
- 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. Kompiuterinė 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* 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 programinės įrangos kūrimo metu, 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*, autoriai akcentuoja, kad gerai įgyvendintos, objektiškai orientuotos sistemos turėtų vystytis be didelių pakeitimų jų architektūroje. To siekiama todėl, nes architektūriniai pakeitimai paveikia didelę sistemos dalį ir jų įgyvendinimo ir priežiūros kaštai yra žymiai didesni[Sho19]. Paketų struktūra, kuri užtikrina atsietą (angl. *decoupled*) komunikavimą tarp paketų, 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 paketais 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 reikia, 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* 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 metrikos

Beveik visi autoriaus aprašyti principai turi aiškiai apibrėžtas metrikas, kuriomis galima pamatuoti, kaip stipriai paketas laikosi šių principų. Būtent šios metrikos bus naudojamos įvertinti paketus analizuojamuose šablonuose norint patikrinti šablonų kokybę:

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

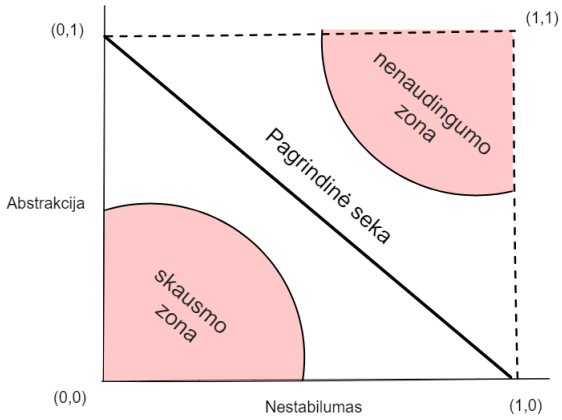
$$Nestabilumas = \frac{Jungtys_{eferentines}}{Jungtys_{eferentines} + Jungtys_{aferentines}} \quad (1)$$

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

$$Abstrakcija = \frac{N_{abstrakcios}}{N_{visos}} \quad (2)$$

  Ši metrika 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** - 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 (Nestabilumas=0, Abstrakcija=1) arba visiškai konkretūs ir stabilūs (Nestabilumas=1, Abstrakcija=0).




1 pav. Pagrindinės sekos kreivė

Atstumą nuo pagrindinės sekos galima apskaičiuoti kaip:

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

Ši metrika yra paketo abstraktumo ir stabilumo pusiausvyros rodiklis. Šios metrikos diapazonas yra nuo nulio iki vieno, kur nulis reiškia paketą, sutampantį su pagrindine seka, o vienas - paketą, maksimaliai nutolusį nuo pagrindinės sekos.

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

Šių metrikų patikimumas buvo ivertintas atvejo analizėje *Exploring the Relationships between Design Metrics and Package Understandability: A Case Study*, kurioje autoriai tyrinėjo sąryšį tarp minėtų metrikų 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 metrikų ir paketų suprantamumo [Eli10].

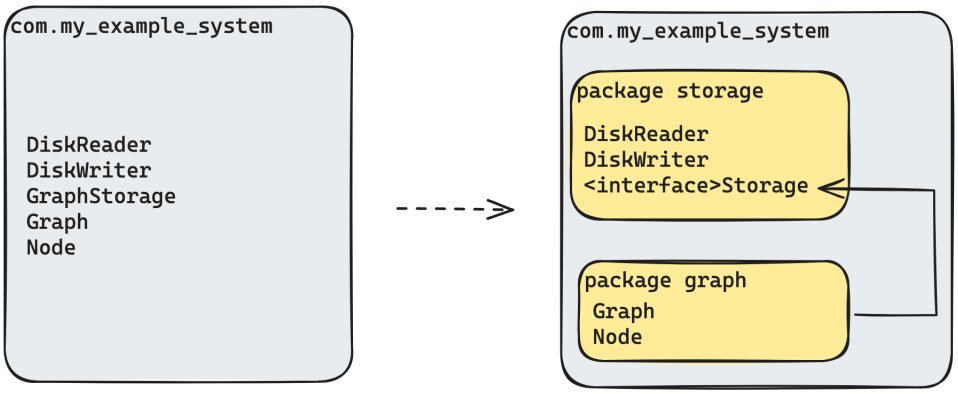
# 2. Galimi kodo skirstymo į paketus šablonai

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 šiuos šablonus, buvo nagrinėjamos atviro kodo sistemos, stebimi nukrypimai nuo numatytojo skirstymo metodo bei ieškoma paaiškinimų, kodėl šie sprendimai buvo priimti. Nuspręsti, kaip žmonės supranta programinį kodą yra gan sudėtingas ir subjektyvus procesas, todėl aprašant šablonus kodo skirstymui geriau akcentuoti, kaip sugrupuoti paketai bendrauja tarpusavyje ir skirstyti juos pagal klasių naudojimo atvejus ir priklausomybes. Taip kodo grupavimo metodai yra labiau artimi Martino aprašytiems principams.

## 2.1. Pagal komponentą

Organizavimas pagal komponentus sumažina sistemos sudėtingumą, pabrėždamas išorinę ir vidinę kodo vienetų darną. Išorinė darna reiškia, kad paketas turi minimalią sąsają (angl. *interface*), kuri atskleidžia tik konceptus (metodus arba duomenų tipus), kurie yra glaudžiai susiję su komponento teikiama paslauga. Vidinė darna reiškia, kad pakuotėje esantis kodas yra stipriai susijęs tarpusavyje ir susijęs su teikiama paslauga.

Kodas yra grupuojamas į mažus paketus, turinčius vieną, aiškiai apibrėžtą funkcionalumą ar tikslą, aprašant abstrakciją, kokie paketo elementai yra pasiekiami iš išorės ir kaip jie naudojami. Taip sukuriamas kodas, kuris yra lengviau suprantamas. Tokią kodo grupavimo tvarką sunku palaikyti, tačiau jos rezultatas – kodas, kuris yra lengviau suprantamas, lengviau pagerinamas, lengviau testuojamas ir, dėl aiškiai aprašytų sąsajų, lengviau pernaudojamas.




2 pav. Sistemos sugrupuotos pagal komponentą pavyzdys

## 2.2. Problemos

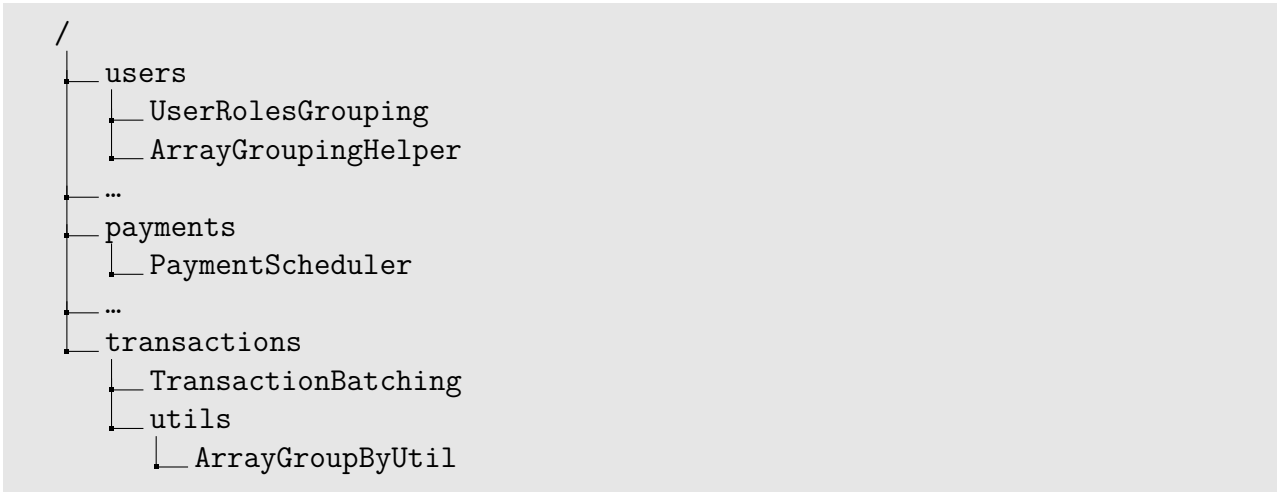
Norint išskirti galimus šablonus paketams skirstyti, reikėtų analizuoti jų panaudojimą atviro kodo sistemose, siekant pastebėti, kur buvo nukrypta nuo bendresnio kodo skirstymo būdo ir identifikuoti klausimus arba problemas, kurias buvo bandoma išspresti.

| Paketo vardas | $N$ | $A$ | $E$ | $S$ | $A$ | $D$ |
|---|---|---|---|---|---|---|
| com.google.inject.internal.util | 8 | 2 | 13 | 0.867 | 0.125 | 0.008 |
| com.google.inject | 29 | 7 | 16 | 0.696 | 0.345 | 0.041 |
| com.google.inject.spi | 52 | 4 | 27 | 0.871 | 0.538 | 0.409 |
| com.google.inject.matcher | 4 | 3 | 4 | 0.571 | 0.5 | 0.071 |
| com.google.inject.multibindings | 15 | 2 | 14 | 0.875 | 0.267 | 0.142 |
| com.google.inject.internal | 116 | 6 | 52 | 0.897 | 0.172 | 0.069 |
| com.google.inject.util | 5 | 4 | 12 | 0.75 | 0.0 | 0.25 |
| com.google.inject.binder | 7 | 4 | 3 | 0.429 | 0.857 | 0.286 |
| com.google.inject.name | 4 | 1 | 7 | 0.875 | 0.0 | 0.125 |
| com.google.inject.internal.aop | 17 | 1 | 22 | 0.957 | 0.118 | 0.075 |

| $\bar{N}$ | $\bar{A}$ | $\bar{E}$ | $\bar{S}$ | $\bar{A}$ | $\bar{D}$ |
|---|---|---|---|---|---|
| 26 | 3 | 17 | 0.779 | 0.292 | 0.148 |

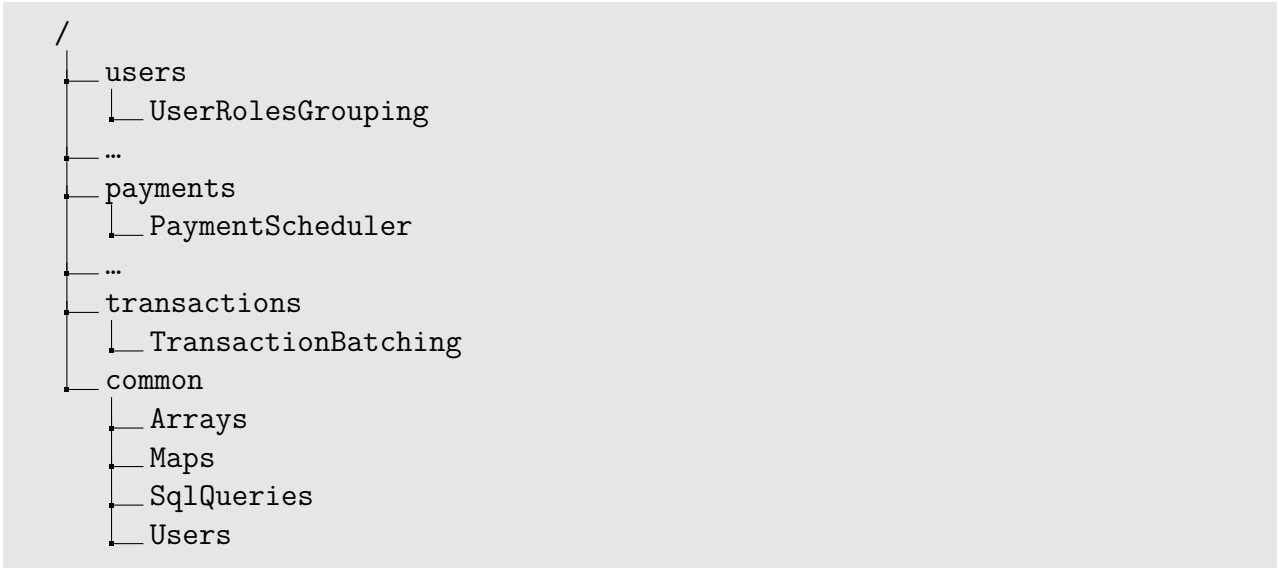
### 2.2.1. Pagalbinių, daugkartinio naudojimo klasių skirstymas

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.




3 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.




4 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. Tokį pagalbinių klasių skirstymo metodą naudoja keletas repozitorijų - pavyzdžiui, užrašinės aplikacijos [1] bei [2]

```
/common
├── arrays
│   ├── ArrayFilters
│   └── ArrayComparators
├── …
├── maps
│   ├── MapTransformations
│   └── MapJoining
├── json
│   └── JsonParser
├── …
└── database
    ├── DatabaseConnection
    └── DatabaseQueries
```

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

Naudojant tokį šabloną, programuotojas, susiduriantis su bendrine problema, kuri, labai tikėtina, jau yra išspresta 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.

[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`

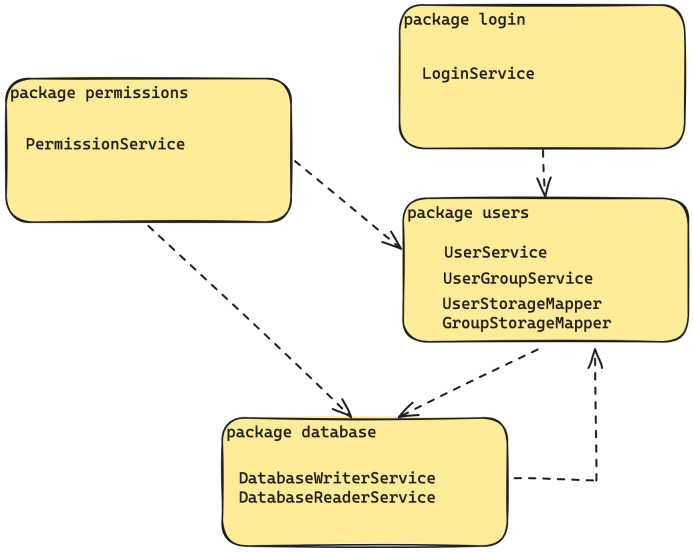
Robert C. Martin bendro sąryšio principas, kuris teigia, kad visos tarpusavyje susijusios klasės turėtų būti vienam pakete, akcentuoja siekiamybę turėti gan mažus paketus, turinčius aiškiai apibrėžtą funkcionalumą, priežastį egzistuoti, taip užtikrinant glaudų tarpusavyje susijusių klasių saryšį. Šis principas galėtų būti kodo skirstymo šablonas, užtikrinantis racionalų aferentinių jungčių skaičių paketuose. Vadovaujantis šiuo šablonu kiekvieną paketą reikėtų realizuoti kaip komponentą, teikiantį vieną funkcionalumą, turintį minimalią sąsają (angl. *interface*), kuri atskleidžia tik konceptus (metodus arba duomenų tipus), kurie yra glaudžiai susiję su komponento teikiama paslauga.

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

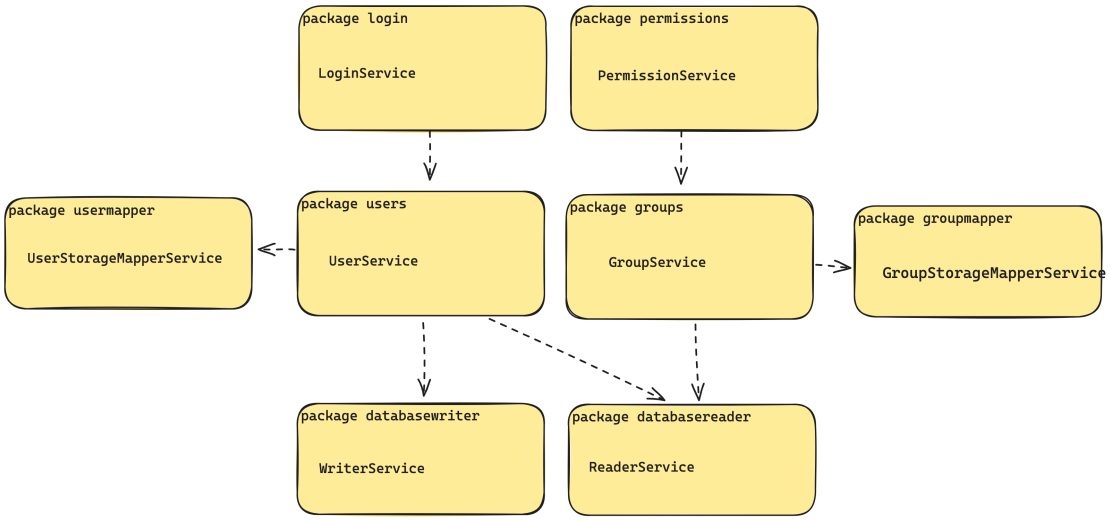
Taip pat mažas paketo funkcionalumas reiškia, kad minėtas paketas skirtas funkcionalumui įgyvendinti naudos minimalų kitų sistemos esybių skaičių, taip sumažinant ir eferentinių jungčių skaičių.

Toks skirstymo būdas

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




6 pav. Sistemos pavyzdys su kelias funkcijas atliekančiais paketais




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

### 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 sub paketus kiekvienos sąsajos implementacijoms. Taip pat, jei vienos sąsajos implementacijai reikalingos kelios klasės, jas galima išskirti į atskirą paketą.

### 2.2.4. Ka daryti su mikroservisų architektūra?

### 2.2.5. Ka daryti su paslėptom priklausomybėm?

### 2.2.6. Kaip grupuoti kodą mono repozitorijoj?

### 2.2.7. Ka daryti su greitai besikeičiančiu kodu?

### 2.2.8. 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 bei iškeliant bendras priklausomybes taip, kad jas galėtų pasiekti abi versijos. Taip galima išvengti kodo duplikacijos bei perteklinio klasių skaičiaus paketuose.

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

8 pav. Sistemos pavyzdys, kurioje skirtingos versijos patalpintos atskiruose sub paketuose, paliekant tik bendro naudojimo klases

Toks skirstymo būdas matomas keliose repozitorijose - Mongo duomenų bazėje[3], API skirtame kelionių valdymui[4] bei duomenų saugojimo įrankyje [5]. Šiose repozitorijose v1, v2 pavadinti paketai saugo skirtingas esybių versijas.

Packaging microservice classes involves structuring your codebase in a way that promotes modularity, scalability, and maintainability within the microservices architecture. Here's how you can package microservice classes effectively:

2. *Separate Concerns*: - Follow the single responsibility principle (SRP) by separating concerns within each package. - For example, separate classes responsible for handling HTTP requests, business logic, data access, and external integrations into distinct packages or modules.

5. *API Contract Packaging*: - Define clear API contracts for your microservices, specifying the expected inputs, outputs, and behavior of each service. - Group classes related to API endpoints, request/response models, and error handling into dedicated packages representing the API contract.

6. *Infrastructure Concerns*: - Separate infrastructure-related concerns, such as database access, caching, logging, and messaging, into their own packages or modules. - This helps isolate infrastructure-specific code and configurations, making it easier to maintain and evolve over time.

code duplication

8. *Testing Concerns*: - Organize test classes alongside production code, following a similar package structure. - Group unit tests, integration tests, and end-to-end tests into separate packages or modules to maintain clarity and organization.

3. *Difficulty in Understanding and Navigation*: - Unorganized code makes it hard for developers to understand the structure and relationships between different parts of the system. - Lack of clear packaging boundaries makes navigation and code exploration challenging, especially for new team members or maintainers.

To solve the problem of multiple interface implementations, you can use several packaging techniques. Here are some approaches:

1. *Interface Segregation Principle (ISP)*: - Follow the ISP, one of the SOLID principles, by segregating interfaces based on client requirements. - Instead of having a single interface with

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

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

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

multiple methods, create smaller, more focused interfaces with a single responsibility. - Group related methods into separate interfaces, allowing classes to implement only the interfaces that are relevant to them.

2. *Package by Feature*: - Organize classes implementing related interfaces into packages based on feature or functionality. - Each package contains classes that provide a specific feature and implement the necessary interfaces for that feature. - This approach keeps related classes together and reduces the likelihood of conflicts between different interface implementations.

5. *Factory Pattern*: - Use the factory pattern to create instances of classes implementing specific interfaces. - Define factory methods or classes that encapsulate the logic for instantiating different implementations of an interface. - This centralizes the creation logic and allows for dynamic selection of implementations based on runtime conditions.

6. *Composition over Inheritance*: - Prefer composition over inheritance when designing class relationships. - Instead of implementing multiple interfaces directly in a single class, consider composing the class of multiple smaller components, each implementing a specific interface. - This promotes flexibility, as you can mix and match components to achieve different behaviors without creating complex inheritance hierarchies.

1. *Modularity*: Packaging allows breaking down a large system into smaller, more manageable modules. This enhances modularity, making it easier to understand, maintain, and extend the codebase.

4. *Code Reusability*: Well-packaged code promotes code reuse by encapsulating reusable components or libraries into separate packages. This reduces duplication and improves overall productivity.

6. *Collaboration*: Packaging encourages collaboration among team members by providing clear boundaries between different parts of the system. It enables parallel development and reduces the risk of conflicts when multiple developers work on the same project.

Could you give patterns how to group code into packages:

Certainly! Grouping code into packages is a common practice in many programming languages to organize and manage large codebases effectively. Here are some patterns for grouping code into packages:

1. *Functional Grouping*: - Group code based on functionality or domain. For example, in a web application, you might have packages like authentication, user-management, payment, etc. - Each package contains related modules, classes, or functions that handle specific tasks within that domain.

2. *Layered Architecture*: - Organize code into layers such as presentation, business logic, and data access layers. - Each layer represents a different aspect of the application, with well-defined responsibilities and dependencies.

3. *Feature-Based Packaging*: - Group code based on features or components of the application. - Each feature or component has its own package containing all the necessary code, including UI, logic, and data handling.

4. *Namespace Packaging*: - Use namespaces or packages to organize code based on

hierarchical relationships. - For example, in Java, you might have packages like com.example.project.module1, com.example.project.module2, etc.

5. *Modular Packaging*: - Divide the application into smaller, independent modules that can be developed, tested, and deployed separately. - Each module may contain its own set of packages, making it easier to manage dependencies and promote code reuse.

6. *Aspect-Oriented Packaging*: - Separate cross-cutting concerns such as logging, security, and caching into their own packages. - These packages can then be applied to different parts of the application as needed, using techniques like aspect-oriented programming (AOP).

7. *Dependency Injection Packaging*: - Organize code based on the dependency injection (DI) hierarchy. - Group classes and components that depend on each other closely within the same package to facilitate DI configuration and management.

8. *Component-Based Packaging*: - Treat each component of the application as a standalone unit with its own package. - Components may include UI widgets, services, data models, etc., each residing in its own package for easier maintenance and reuse.

By applying these packaging patterns, you can create a well-organized codebase that is easier to understand, maintain, and extend over time

# 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 kokybes metrikas yra sudėtingas procesas. Daryti tai rankiniu budu 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. Šiame darbe nagrinėjamos Java programavimo kalba parašytos sistemos.

## 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 metrikų skaičiavimas – įrankis ar įrankiai turi padeda apskaičiuoti aprašytas paketų kokybės metrikas. 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 kokybėi 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 metrikas:

- 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 metrikos, bei šių metrikų 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. Kodo skirstymo metodai realiose sistemose

## 4.1. Sistemų pasirinkimas

## 4.2. Sistemų analizės procesas

# 5. Sistemų pertvarkymas pagal šablonus

# 6. Medžiagos darbo tema dėstymo skyriai

Medžiagos darbo tema dėstymo skyriuose išsamiai pateikiamos nagrinėjamos temos detalės: pradiniai duomenys, jų analizės ir apdorojimo metodai, sprendimų įgyvendinimas, gautų rezultatų apibendrinimas.

Medžiaga turi būti dėstoma aiškiai, pateikiant argumentus. Tekste dėstomas trečiuoju asmeniu, t.y. rašoma ne „aš manau“, bet „autorius mano“, „autoriaus nuomone“. Reikėtų vengti informacijos nesuteikiančių frazių, pvz., „...kaip jau buvo minėta...“, „...kaip visiems žinoma...“ ir pan., vengti grožinės literatūros ar publicistinio stiliaus, gausių metaforų ar panašių meninės išraiškos priemonių.

Skyriai gali turėti poskyrius ir smulkesnes sudėtines dalis, kaip punktus ir papunkčius.

## 6.1. Poskyris

Citavimo pavyzdžiai: cituojamas vienas šaltinis [**PvzStraipsnLt**]; cituojami keli šaltiniai [**PvzStraipsnEn**; **PvzStraipsnLta**; **PvzKonfLt**; **PvzKonfEn**; **PvzKnygLt**; **PvzKnygEn**; **PvzElPubLt**; **PvzElPubEn**; **PvzBakLt**; **PvzMagistrLt**; **PvzPhdEn**].

Anglų kalbos terminų pateikimo pavyzdžiai: priklausomybių injekcija (angl. *dependency injection*, dažnai trumpinama kaip *DI*), saitų redaktorius (angl. *linker*).

Išnašų[6] pavyzdžiai[7].

## 6.2. Faktorialo algoritmas

1 algoritmas parodo, kaip suskaičiuoti skaičiaus faktorialą.

**1 algoritmas.** Skaičiaus faktorialas

1: $N \leftarrow$ skaičius, kurio faktorialą skaičiuojame
2: $F \leftarrow 1$
3: **for** $i := 2$ to $N$ **do**
4: $\quad F \leftarrow F \cdot i$
5: **end for**

### 6.2.1. Punktas

#### 6.2.1.1. Papunktis

### 6.2.2. Punktas

[6] Pirma išnaša.

[7] Antra išnaša.

# 7. Skyrius

## 7.1. Poskyris

## 7.2. Poskyris

# 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


[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~doSlc6JoeK7XXJ5bGLPB1B1yBDOtnojJ0yNuWZzQP9fpTjd~yffOhlxnM4GyB2gNMuGZyXsDBXWQuD66kZpwWdluJ63GWjvs28T2ArRpaekqk9JAc-Icun18nyonYJ~W4pIViibjHA4k9yN5r1FZRSWFeUSHpRmflEpJ1opD2Nh889TquLxsA2DhPP3L5_&Key-Pair-Id=APKAJLOHF5GGSLRBV4ZA.

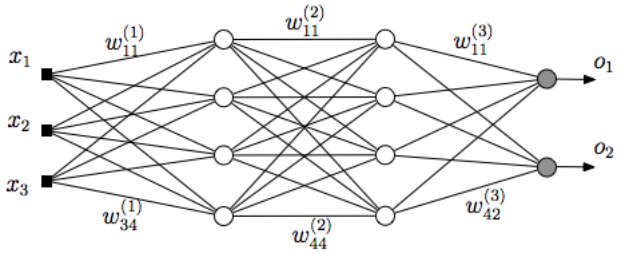
# Santrumpos

Sąvokų apibrėžimai ir santrumpų sąrašas sudaromas tada, kai darbo tekste vartojami specialūs paaiškinimo reikalaujantys terminai ir rečiau sutinkamos santrumpos.

# Priedai

## Priedas nr. 1

## Neuroninio tinklo struktūra




9 pav. Paveikslėlio pavyzdys

## Priedas nr. 2

## Eksperimentinio palyginimo rezultatai

1 lentelė. Lentelės pavyzdys

| Algoritmas | $\bar{x}$ | $\sigma^2$ |
|---|---|---|
| Algoritmas A | 1.6335 | 0.5584 |
| Algoritmas B | 1.7395 | 0.5647 |