

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. Teisingai įgyvendinta kompiuterinė sistema	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. Pagal techninį sluoksnį	13
2.3. Pagal tipą	13
2.4. Pagal funkciją	14
2.5. Problemos	14
2.5.1. Kur laikyti pagalbines klases, daugkartinio naudojimo klases?	14
2.5.2. Ka daryti esant dideliame priklausomybių nuo paketo skaičiui?	16
2.5.3. Ka daryti su ciklinėmis priklausomybėmis?	17
2.5.4. Ka daryti su daug skirtingų sąsajų implementacijų?	17
2.5.5. Ka daryti su mikroservisų architektūra?	17
2.5.6. Ka daryti su pasleptom priklausomybėmis?	17
2.5.7. Kaip grpuoti kodą mono repozitorijoj?	17
2.5.8. Ka daryti greitai besikeičiančių kodu?	17
2.5.9. Ka valdyti esybių evoliuciją ir versijavimą	17
3. ĮRANKIAI ŠABLONŲ ANALIZEI IR ĮVERTINIMUI	20
3.1. Reikalavimai įrankiams	20
3.2. Reikalavimai įrankiui bendrinės sistemos analizei	20
3.3. Reikalavimai įrankiui paketo kokybei skaičiuoti	21
3.4. Įrankių įgyvendinimas	21
4. KODO SKIRSTYMO METODAI REALIOSE SISTEMOSE	22
4.1. Sistemų pasirinkimas	22
4.2. Sistemų analizės procesas	22
5. SISTEMŲ PERTVARKYMAS PAGAL ŠABLONUS	23
6. MEDŽIAGOS DARBO TEMA DĖSTYMO SKYRIAI	24
6.1. Poskyris	24
6.2. Faktorialo algoritmas	24
6.2.1. Punktas	24
6.2.1.1. Papunktis	24
6.2.2. Punktas	24
7. SKYRIUS	25
7.1. Poskyris	25
7.2. Poskyris	25
REZULTATAI	26

IŠVADOS	27
ŠALTINIAI	28
SANTRUMPOS	29
PRIEDAI	30
1 priedas. Neuroninio tinklo struktūra	30
2 priedas. Eksperimentinio palyginimo rezultatai	31

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 studijavimas ir analizė norint įvertinti programinės įrangos kokybę tampa vis svarbesne tema dėl augančio failų ir paketų skaičiaus[Eli10].

Norint išsiaiškinti, kaip efektyviausiai gali būti skaidomas programinis kodas, tam jog jo struktūra darytų teigiama 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, užtikrinant nuoseklų stilių.

Šio darbo tikslas – identifikuoti ir įvertinti šablonus kodo skirstymui į paketus. Remiantis moksliniais straipsniais apie sistemos kokybę bei palaikomumą aprašyti kriterijus, kurie būtų naudojami šablonus įvertinti, nustatant jų įtaką sistemos palaikomumui.

Tikslui pasiekti yra iškeliama 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 rastai gerai įgyvendintos sistemos požymiais
- Pasirinkti kelias sistemas ir pertvarkyti jų failų struktūrą pagal aprašytus šablonus, įvertinant kiek sudėtinga pasiekti kiekvieno šablono struktūrą
- Naudojant pertvarkytas sistemas, įvertinti kiekvieną kodo skirstymo šabloną pagal pasiūlytus kriterijus
- Pateikti rekomendacijas, kokius šablonus kodo skirstymui tinkamiausia naudoti

Šiuo 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. Pasirintos ir išnagrinėtos 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 struktūros 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 šablonų klases 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 laiktųsi 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. Teisingai įgyvendinta kompiuterinė sistema

Norint išsiaiškinti, kokią įtaką sistemos kokybei daro skirtingos paketų skirstymo metodologijos ir kaip objektyviai pamatuoti jų įtaką, pirmiausia reikėtų apibrėžti, kokiais požymiais pasižymi teisingai 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 kompleksiskumas. 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 turėtų 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. Programinės įrangos niuansas, kad modulis, sukurtas taip, kad jį būtų lengva pakeisti, kartais tampa sunkiai keičiamu, kažkam pridėjus priklausomybę nuo jo[Mar02].

1.2.4. Stabilių abstrakcijų principas

Paketas 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 dizainui[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 pasirinktame 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 stabilų priklausomybių principu:

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

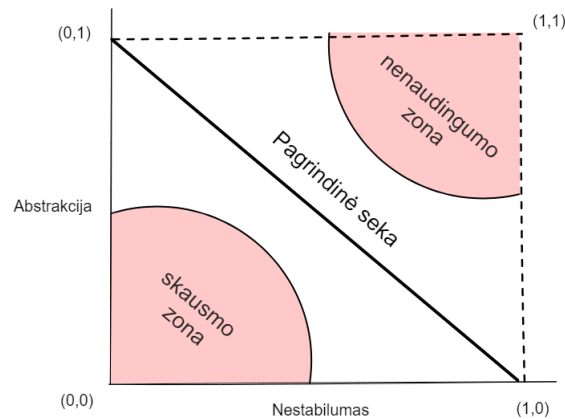
Reikšmės režiai – nuo nulio iki vieno, kur vienas nurodo visiškai stabilų paketą, o vienetą – visiškai nestabilų.

- **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 vienetą nurodo, kad pakete yra tik abstrakčios klasės.

- **Atstumas nuo pagrindinės sekos** – Pagrindinė seka tai sąryšis tarp nestabilumo ir abstrakcijos. Pagrindinė seka yra idealizuota linija ($\text{Abstrakcija} + \text{Nestabilumas} = 1$) kurią galima atvaizduoti 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 ($\text{Nestabilumas}=0$, $\text{Abstrakcija}=1$) arba visiškai konkretūs ir stabilūs ($\text{Nestabilumas}=1$, $\text{Abstrakcija}=0$).



1 pav. Pagrindinės sekos kreivė

Atstumą nuo pagrindinės sekos galima apskaičiuoti kaip:

$$\text{Atstumas} = |\text{Abstrakcija} + \text{Nestabilumas} - 1| \quad (3)$$

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 principu.

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 programuotojus. Jų buvo paprašta pilnai suprasti paketų funkcionalumą ir nuo vieno iki dešimt įvertinti pastangas, reikalingas suprasti kiekvieną paketą. Rezultatai gauti iš šio tyrimo rodo statistškai reikšmingą koreliaciją tarp daugumos metrikų ir paketų suprantamumo[Eli10].

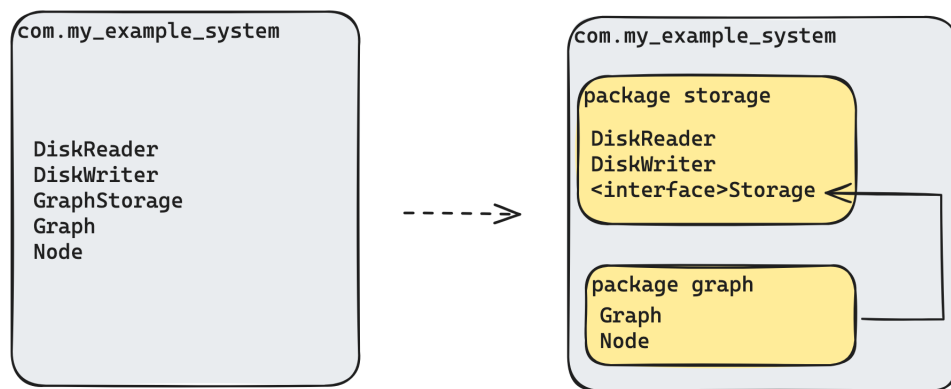
2. Galimi kodo skirstymo į paketus šablonai

Diskusijose, kaip reikėtų skirstyti programinį kodą, paprastai akcentuojami du šablonai – pagal *techninį sluoksnį*, todo: šaltiniai, nurodantys skirstymo būdus 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. Tačiau šie du šablonai yra gan platūs ir galėtų būti išskaidyti į daugiau smulkesnių ir tiksliau aprašytų šablonų. Taip pat, minėtuose šablonuose, būdai kaip ir kodėl skaidyti programinį kodą parinkti akcentuojant tai, kaip programinį kodą supranta žmonės, dirbantys prie to kodo. Nuspresti, 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. Šablonus kaip grupuoti kodą, akcentuojant klasių naudojimo atvejus ir priklausomybes nagrinėja Martin Sandin savo straipsnyje *Four Strategies for Organizing Code*. Šis straipsnis idomus tuo, kad autorius nesiplečia į du dažniausiai sutinkamus šablonus – grupuoti pagal techninį sluoksnį arba dalykinės srities esybes, o aprašo keturis grupavimo būdus arba šablonus, kurie, nors ir įkvėpti minėtų dviejų būdų, yra gan unikalūs ir labiau techniškai apibrėžti.

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. Pagal techninį sluoksnį

Laikantis skirstymo pagal techninį sluoksnį, kiekvienam funkcionalumui arba kompiuterinės sistemos sluoksniui yra sukuriamas paketas, kuris savyje grupuoja skirtingas dalykinės srities esybes. Pavyzdžiui, visos sąsajos darbai su duomenų baze guli viename pakete, sąsajos su verslo logikos transformacijomis kitame, o duomenų vaizdavimo klientui logika trečiame pakete. Šis paketų skirstymo metodas yra labai paplitęs, ypač tarp senesnių kompiuterinių sistemų. Jį paprasta įgyvendinti, metode paprasta pavadinti paketus, aišku, į kuriuos paketus priskirti klases. Nors tokią kodo struktūrą lengva įgyvendinti ir palaikyti sistemoje, ji turi nemažai trūkumų. Šis metodas nepalengvina sistemos plėtros valdymo – augant sistemai paprastai daugėja dalykinės srities esybių, o ne funkcinį sluoksnių, todėl esamų paketų skaičius beveik nesikeičia, bet klasių kiekis pakete vis auga, tai daro neigiamą įtaką navigacijai paketo viduje. Skirstant paketus pagal funkciją taip pat nėra gerinamas informacijos slėpimas (inkapsuliacija) – kiekvienas paketas savyje talpina skirtingų dalykinės srities esybių kodą, tai sudaro sąlygas per klaidą užmegzti komunikaciją tarp komponentų, kurie neturėtų būti susiję. Taip pat, skirstant tokiu būdu, sąsajos yra stipresnės tarp loginių komponentų, pasiskirsčiusių per sluoksnius, nei tarp vieno sluoksnio esybių. Tokiu atveju pokyčių pristatymas pasidaro sudėtingas, nes reikalinga keisti ne vieną sluoksnį. Tačiau šis metodas turi ne tik trūkumus – vienas iš jo privalumų – ganėtinai aiškiai nusakoma bendra sistemos architektūra, programiniai sluoksniai. Deja, netvarkinga sistemos būseną eliminuoja ši privalumą.

2.3. Pagal tipą

Kodo organizavimas pagal tipą įprastai nėra griežtai apibrėžtas – klasės grupuojamos pagal vartotojo sumanytą tipą, neteikiant svarbos klasių sąryšiams ar loginėms esybėms. Taip skirstant klases, į paketus galėtų būti grupuojamos esybių, išimčių ar serviso klasės. Šis skirstymo būdas neteikia prioriteto nei skirstymui pagal techninius sluoksnius, nei pagal dalykinės srities esybes. Jį ganėtinai nesudėtinga įgyvendinti, tačiau šis metodas yra netvarkingas bei turi trūkumų. Taip suskirstytas kodas sunkiai skaitomas, nes nėra aišku, pagal kokią tvarką ieškoti konkrečios klasės ar kaip priskirti esamiems paketams netinkamas klases. Taip pat toks skirstymo būdas visiškai

nepadeda spręsti klasių sąsajų problemų, kadangi neapgalvotai išskirstyti sluoksniai gali būti glaudžiai susiję.

2.4. Pagal funkciją

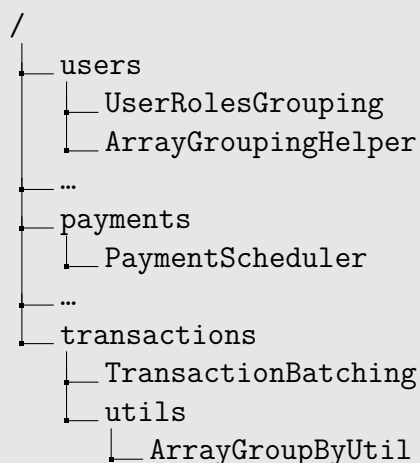
Šis skirstymo būdas dalinai panašus į skirstymą pagal komponentą, tačiau yra mažiau griežtas ir prioritetas teikiamas ne vidinei darnai ir glaudžiai grupuojamų komponentų sąsajai, o išorinei darnai. Tokio skirstymo būdo paketuose dažniausiai grupuojamos tos pačios sąsajos implementacijos, parenkamos siekiant pabrėžti išorinę darną ir sugrupuoti klases, teikiančias panašias funkcijas. Toks skirstymas patogus, pavyzdžiui, techninių bibliotekų vartotojams, kadangi galima lengvai surasti bibliotekos teikiamas funkcijas. Tokią kodo grupavimo tvarką sunku palaikyti, nes reikia gerai apgalvoti skirstymo strategiją, kad ji būtų prasminga ir patogi naudoti.

2.5. Problemos

Aprašant galimus šablonus paketams skirstyti, reikėtų identifikuoti klausimus arba problemas, kurias išspręsti gali gerai sugrupuotas kodas, arba pristatyti / sustiprinti kodas sugrupuotas neteisingai.

2.5.1. Kur laikyti pagalbines klases, daugkartinio naudojimo klases?

Didžiausia problema susijusi su pagalbėmis 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 pavidys, kur labai panšia funkcija atliekančios klasė *ArrayGroupByUtil* ir *ArrayGroupingHelper* egzistuoja todėl, kad inžinierius nerado jau įgyvendintos klasės, dėl aiškos struktūros, daugartinio panaudojimo klasėmis, trūkumo.

Vienas iš šablonų sprendžiančių šią problemą, turėti vieną paketą, skirtą visoms pagalbėmis klasėmis, kuris yra paminėtas sistemos dokumentacijoje ir apie jo egzistavimą teoriškai žino

visi komandos nariai. Pakete turėtų atskiras klases, kiekvienam bendriniam domenui, iš kurios pavadinimo programuotojas galėtų nuspresti, kad jo ieškomas funkcionalumas, bus butent toje klasėje.

```
/
├── users
│   └── UserRolesGrouping
├── ...
├── payments
│   └── PaymentScheduler
├── ...
├── transactions
│   └── TransactionBatching
└── common
    ├── Arrays
    ├── Maps
    ├── SqlQueries
    └── Users
```

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

Jei pagalbinių klasių dydis labai išauga, galima jas 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 subdomena padengia būtent ta klasė.

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

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

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, skirta bendrinio panaudojimo kodui
2. Pakete surasti klasę, kurios pavadinimas būtų susijęs su jo problema
3. Klasės funkcijų sarašę 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. Iškvieisti rastą arba sukurtą funkciją iš bendrinio panaudojimo kodo paketo savo funkcionalume

2.5.2. Ka daryti esant dideliam priklausomybių nuo paketo skaičiui?

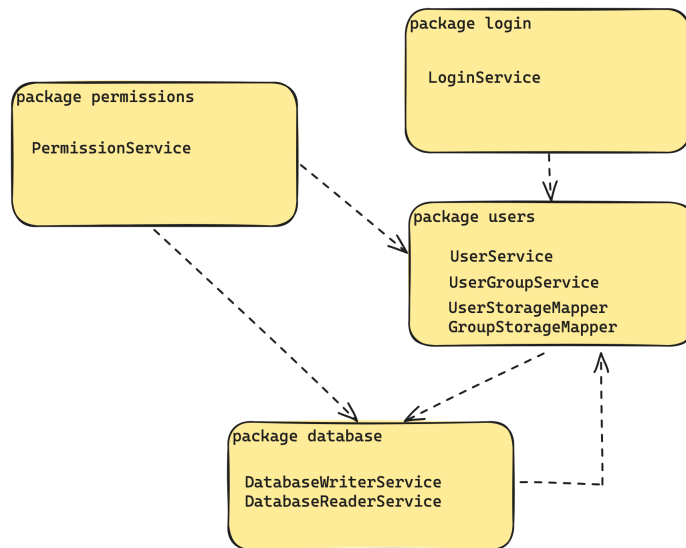
Didelis priklausomybių nuo specifinio paketo skaičius (arba aferentinės jungtys), reiškia, kad pokyčiai tame pakete turės įtaką kelioms klasėms, jei tokia tendencija yra būdinga visai sistemai, sistema tampa mažiau lanksti pokyčiams, kadangi net ir paprastas pakeitimas daro įtaką reišmingai sistemos daliai, pokyčiai yra labiau linkę keisti bendra sistemos architektūrą. Taip pat naujos sistemos versijos išleidimo (angl. *release*) procesas tampa sudėtingis, 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, aiškiai apibrėžtą funkcionalumą, priežastį egzistuoti, taip užtikrinant glaudų tarpusavyje susijusių klasių sąryšį. Šis principas galėtų būti kodo skirstymo šablonas, užtikrinantis racionalų aferentinės jungčių skaičių paketuose. Vadovaujantis šiuo šablonu kiekvieną paketą reikėtų realizuoti kaip komponentą, teikianti vieną funkcionalumą. Turinti minimalią sąsają (angl. *interface*), kuri atskleidžia tik konceptus (metodus arba duomenų tipus), kurie yra glaudžiai susiję su komponento teikiama paslauga.

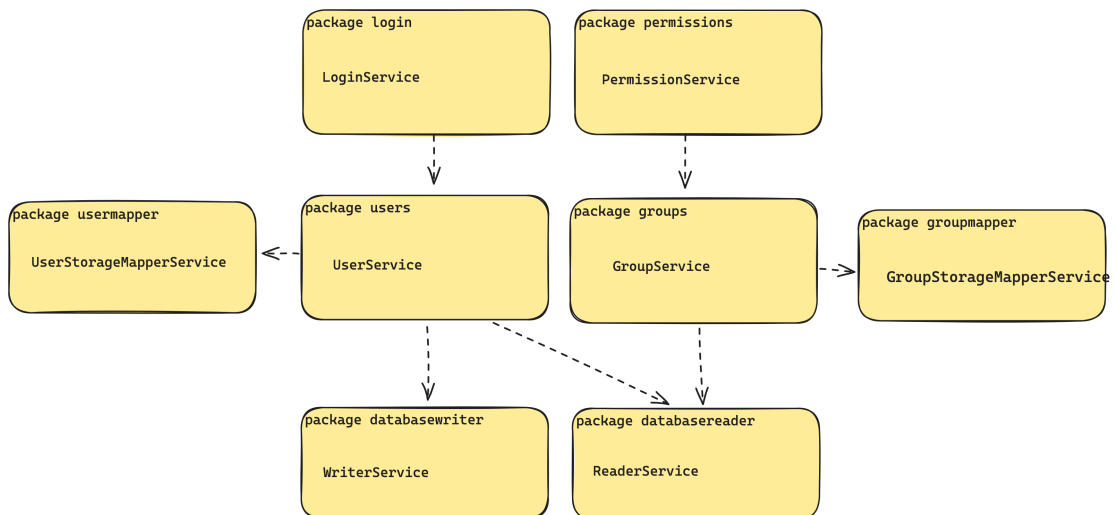
Paketas turintis vieną funkciją yra naudojamas tik tų paketų, kuriems reikia būtent tos funkcijos, taip užtikrinant, tik maža sistemos dalies priklausomybę nuo to vieno paketo.

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

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



6 pav. Sistemos pavizdys su kelias funkcijas atliekančiais paketais



7 pav. Sistemos pavizdys su aiškia, vieną funkciją turinčiais paketais

2.5.3. Ka daryti su ciklinėmis priklausomybėmis?

2.5.4. Ka daryti su daug skirtingų sąsajų implementacijų?

2.5.5. Ka daryti su mikroservisų architektūra?

2.5.6. Ka daryti su pasleptom priklausomybėmis?

2.5.7. Kaip grpuoti kodą mono repozitorijoje?

2.5.8. Ka daryti greitai besikeičiančių kodu?

2.5.9. Ka valdyti esybių evoliuciją ir versijavimą

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 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 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, atsakomybės 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 atvaizduota, meta informacija apie sistema – paketų struktūrą, jų priklausomybes, figuruojančių paketų bei klasių vardai. Ši papildoma informacija nėra aiškūs teiginiai, o tik pagalba analizę atliekančiams asmeniui, leidžianti priimti išvalgas apie sistemą, kaip pavyzdžiui, kokiam paketų skirstymo šablonui yra atimčiausia sistemos struktūra, arba kaip lengvai sistema yra suprantama.
- Paketų kokybės metrikų skaičiavimas – įrankis ar įrankiai turi padeda paskaičiuoti aprašytas paketų kokybes metrikas. Šių įrankių išvestis – tikslūs, formulėmis pagrįstų skaičiavimų rezultatai apie paketų kokybę, kuriuos galima lyginti tarpusavyje.

todo: akcentuoti kad visas darbas vyksta su java sistemomis

3.2. Reikalavimai įrankiui bendrinės sistemos analizei

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
- Vidutinį klasių pakete skaičiaus
- Paketų ir klasių medį, identifikuojanti abstrakčias klases ar sąsajas
- Paketų priklausomybių grafiką

Gautą rezultatą išvesti suprantamu formatu, leidžiant vartotojui priimti 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 metrikas:

- Klasų skaičius
- Aferentinių jungčių skaičius
- Eferentinių jungčių skaičius
- Nestabilumo santykis
- Abstrakcijos santykis
- Atstumo nuo pagrindinės sekos santykis
- Žiedinių priklausomybių skaičius

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 analizes rezultatus pateikti tolesniame šio dokumento tūrinyje.

3.4. Įrankių įgyvendinimas

Nors beveik visiems reikalavimuose minimiems funkcionalumas galima rasti jau sukurti įrankių, greit ir efektyviai pritaikyti visus juos skirtingoms sistemoms (arba posistemėms) nėra patogų – kiekvieną įrankį reikėtų vykdyti atskirai, su skirtingais vykdymo procesais ir argumentais, taip išvestų rezultatų formatai yra skirtingi. Todėl, norint palengvinti šį procesą – suvienodinti procesų vykdymą, bei gautus rezultatus, visi įrankiai reikalingi analizei, įgyvendinti kaip viena programinė sistema, kuri apdoroja failus nurodytoje sistemos direktorijoje, nuskaitytą *java* failų tūrinį ir sukonstruoja informaciją apie sistemos paketus bei klases. Surinkta informacija tada 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“, „atoriaus 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*).

Išnašų¹ pavyzdžiai².

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:    $F \leftarrow F \cdot i$ 
5: end for
```

6.2.1. Punktas

6.2.1.1. Papunktis

6.2.2. Punktas

¹Pirma išnaša.

²Antra 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 terminais, 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.
- [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=BWU2DSSXDNUjfvT1lUWYspcqN1bW1Fgg~doSlc6JoeK7XXJ5bGLPB1B1yBD0tnojJ0yNuWZzQP9fpTjd~yff0hlxnM4GyB2gNMuGZyXsDBXWQuD66kZpwWdluJ63GWjvs28T2ArRpaekqk9JAc-Icun18nyonYJ~W4pIViibjHA4k9yN5r1FZRSWFeUSHpRmflEpJ1opD2Nh889TquLxsA2DhPP3L5_&Key-Pair-Id=APKAJL0HF5GGSLRBV4ZA.

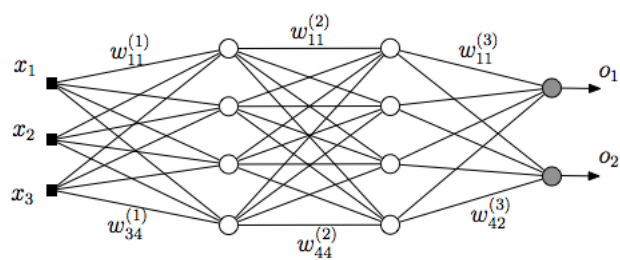
Santrumpos

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



8 pav. Paveikslėlio pavyzdys

Priedas nr. 2

Eksperimentinio palyginimo rezultatai

1 lentelė. Lentelės pavyzdys

Algoritmas	\bar{x}	σ^2
Algoritmas A	1.6335	0.5584
Algoritmas B	1.7395	0.5647