VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
PROGRAMŲ SISTEMŲ STUDIJŲ PROGRAMA

# Kodo skirstymo į paketus būdų analizė ir skirstymo gairių paieška

## Analysis of code division into packages and search for division guidelines

Kursinis darbas

Atliko: 4 kurso 3 grupės studentė
Martyna Ubartaitė

Darbo vadovas: Gediminas Rimša

Vilnius - 2024

# Turinys

# 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. Norint įvertinti, kad sistema įgyvendinta teisingai ir įgalina verslą plėstis, galime pasitelkti Martin Kleppmann knygoje Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems aprašytus 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 komplek-siškumas. 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. 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 ir lengviau palaikomu [jav17]. Modernios sistemos yra didžiulės, programinis kodas yra padalintas į tūkstančius failų, kurie išskaidyti per kelias skirtingo gylio direktorijas ir apgalvotai išskirstytas programinis kodas daro daug didesnę įtaką nei gali atrodyti iš pirmo žvilgsnio. Norint išsiaiškinti, kaip gali būti skaidomas programinis kodas, reikalinga atlikti kiekvieno metodo analizę, suprasti jų privalumus bei trūkumus.

Darbo uždaviniai:

- Įvertinti gerai įgyvendintos sistemos reikalavimus.
- Įvertinti teorinius kodo skirstymo būdus, jų privalumus bei trūkumus.
- Išsiaiškinti, kaip šie būdai naudojami praktikoje.

Šio darbo tikslas yra pateikti gaires, kaip teisingiausiai išskaidyti programos kodą į paketus, jog jis padėtų siekti kriterijų, apibūdinančių gerai įgyvendintą sistemą. Prioritetas turėtų būti skiriamas ne nepriekaištingam, teoriškai geriausiam būdui, o pragmatiškam sprendimui, pasižyminčiam gairių taikymo paprastumu ir potencialu būti naudojamam praktikoje.

# 1. Kodo skirstymo metodų vertinimas

Tvarkingas, aiškiai suprantamas kodas yra subjektyvi tema, priklausanti nuo komandos, naudojamos programavimo kalbos ar programinių įrankių bei programinės sistemos dalykinės srities. Aprašytas idealiausias kodo skirstymo metodas, taip, kaip ir bendros tvarkingo kodo praktikos, gali būti labai subjektyvus ir patogus tik metodą formavusiam asmeniui. Tam, kad būtų pasiektas kuo objektyvesnis, plačiau priimtinas rezultatas, reikėtų aprašyti kriterijus, nurodančius, ko tikimasi iš kodo skirstymo metodo - sprendžiamas programinės įrangos kūrimo problemas, suteikiamas garantijas. Šie kriterijai bus naudojami vertinant skirtingus kodo skirstymo metodus bei aprašant idealiausią skirstymo variantą, taip užtikrinant pagrįstą rezultatą.

## 1.1. Gerai suskirstyto kodo požymiai

Norint užtikrinti, kad aprašyti kriterijai yra ištikrųjų vertingi, turi būti atsižvelgta, kad jie padeda pagerintį sistemos prižiūrimumą, viena iš pagrindinių teisingai įgyvendintos sistemos savybių.

- Informacijos slėpimas (inkapsuliacija) - kodo skirstymas paketais turėtų padėti paslėpti vidinius kodo komponentus ir užtikrinti, jog tik norimi komponentai yra pasiekiami viešai. Tai prisideda gerinant sistemos patikimumą, kadangi tai leidžia paprasčiau modifikuoti programinį kodą ateityje, kiek įmanoma išvengiant nenumatytų pasekmių.
- Sistemos plėtros valdymas - kodas turėtų būti išskaidytas į paketus tokiu būdu, jog augant sistemai, didėjant funkcionalumui, skirstymas padėtų palaikyti sistemą - palengvintų reikalingų komponentų radimą, navigaciją tarp failų, tai supaprastina inžinierių, dirbančių prie sistemos darbą, leidžia greičiau augti sistemai.
- Užuominos į sistemos architektūrą - paketai turėtų padėti suprasti aukšto lygio sistemos architektūrą - koks jos tipas ir kaip skirstomos dalys, kaip jos komunikuoja tarpusavyje. Turint aiškią sistemos architektūros viziją yra lengviau planuoti ateities pakeitimus, nes yra žinoma kokie komponentai turės pasikeisti.
- Užuominos į verslo dalykinę sritį - paketai turėtų padėti suprasti sistemos verslo dalykinę sritį, esamas domeno esybes, pagrindinius verslo funkcionalumus, esybių bendravimą. Tai padeda programuotojams lengviau komunikuoti su sistemos produkto žinovais, suteikti naudingų pasiulymų produkto klausimuose.
- Pagalba prisidedant prie sistemos kūrimo - paketų struktūra lemia, kaip sudėtinga padaryti pakeitimus sistemoje žmogui, nedirbusiam prie jos anksčiau. Šis kriterijus taip pat prisideda gerinant sistemos prižiūrimumą, kadangi gera paketų struktūra palengvina naujų narių integraciją į komandą, o atviro kodo sistemose suteikia galimybę susilaukti daugiau įnašo iš trečiųjų šalių.

# 2. Galimi paketų skirstymo metodai

## 2.1. Pagal techninį sluoksnį

Vienas dažniausiai paplitusių būdų skirstyti kodą yra pagal techninį sluoksnį. Laikantis tokio skirstymo būdo kiekvienam funkcionalumui arba kompiuterinės sistemos sluoksniui yra sukuriamas paketas, kuris savyje grupuoja skirtingas dalykinės srities esybes. Pavyzdžiui, visos sąsajos darbui su duomenų baze guli viename pakete, sąsajos su verslo logikos transformacijomis kitame, o duomenų vaizdavimo klientui logika trečiame pakete.

```
/
├── grafinėssąsajos
│   ├── PrekėsGrafinėSąsaja
│   └── VartotojoGrafinėSąsaja
├── esybės
│   ├── Prekė
│   └── Vartotojas
├── repozitorijos
│   ├── PrekėsRepozitorija
│   └── VartotojoRepozitorija
└── servisai
    ├── PrekėsPirkimoServisas
    └── VartotojoValdymoServias
```

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 daug trūkumų, kuriuos galima pastebėti pritaikius anksčiau aprašytus kriterijus, vertinančius kodo skirstymo būdą. Šis metodas nepalengvina sistemos plėtros valdymo - augant sistemai paprastai daugėja dalykinės srities esybių, o ne funkcinių 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 salygas per klaidą užmegzti komunikaciją tarp komponentų, kurie neturėtų būti susiję.

Kadangi pagrindiniai (angl. *root* paketų pavadinimai nusako techninius sluoksnius, o ne dalykinę sritį, sistemos pirminis kodas nepateikia užtektinai informacijos, užuominų apie sistemos verslo esybes, jų hierarchiją ir kaip jos bendrauja tarpusavyje. Tai taip pat nepalengvina pakeitimų įnešimo į sistemą proceso žmonėms, kurie nėra šios kodo bazės ekspertai, kadangi vienos verslo funkcijos klasės yra išskirstytos per skirtingus paketus ir norint suprasti bendrą funkcionalumo veikimą, reikia perprasti kelių paketų atsakomybes.

Tačiau šis metodas turi ne tik trūkumus - vienas iš jo privalumų - ganėtinai aiškiai nusakoma bendra sistemos architektūra, programiniai sluoksniai. Deja, netvarkinga sistemos būsena eliminuoja ši privalumą. Pavyzdžiui, jei sistemoje figūruoja lavos tėkmės anti modelis (angl. *L*ava Flow Anti Pattern), kuris nusako lavos tėkmes - kodą iš senesnių kūrimo iteracijų, kuris tapo

nejudinamas, nenaudingas, apie kurį neliko informacijos - [Sou], labai tikėtina, kad skirtingos lavos atšakos turės sluoksnių skirtingais pavadinimais, kurie atlieka tą pačia funkciją.

## 2.2. Pagal dalykinės srities esybes

Kodo skirstymo būdas, priešingas skirstymui pagal techninį sluoksnį yra grupavimas į paketus pagal dalykinės srities esybes. Skirstant kodą šiuo metodu vienos esybės kodas, dalykinės srities esybės funkcionalumas, skirtingose programiniuose sluoksniuose yra patalpintas viename pakete.

```
/
├── prekės
│   ├── Prekė
│   ├── PrekėsGrafinėSąsaja
│   ├── PrekėsPirkimoServisas
│   └── PrekėsRepozitorija
├── vartotojai
│   ├── Vartotojas
│   ├── VartotojoGrafinėSąsaja
│   ├── VartotojoRepozitorija
│   └── VartotojoValdymoServias
└── pirkimai
    ├── Pirkimas
    ├── PirkimųGrafinėSąsaja
    ├── PirkimųServisas
    └── IšorinisPirkimųKlientas
```

Šis metodas palengvina sistemos plėtros valdymą augant sistemai - pridedant naujus funkcionalumus, daugėjant paketų skaičiui failų kiekis paketų viduje keičiasi labai retai, dažniausiai tik esant esminiams architektūros pakeitimams. Tokio failų grupavimo dėka net esant labai didelei sistemai, vienos dalykinės srities esybės kodas yra aiškus ir lengvai suprantamas, kadangi jis yra laikomas viename pakete. Tai leidžia pakeisti specifinį sistemos funkcionalumą, nešokinėjant tarp skirtingų paketų net ir sistemoje, sudarytoje iš daug skirtingų modulių. Taip pat, kadangi visi paketai yra laikomi tame pačiame lygyje, o paketų pavadinimai - dalykinės srities esybės, iš projekto paketų struktūros galima suprasti, kokios esybės egzistuoja sistemoje.

Kadangi failai su specifiniais produkto funkcionalumais yra vienoje vietoje, o paketų pavadinimai suteikia užuominų apie produkto dalykinės srities esybes, naujiems komandos nariams arba trečioms šalims (jei tai atviro kodo sistema) yra daug paprasčiau prisijungti prie sistemos kūrimo. Galima prisidėti mažais pakeitimais, neturint bendro sistemos supratimo.

Grupuojant dalykinės srities esybes paketais ir kiekvieną paketą laikant atskiru, save išlaikančiu moduliu, galime užtikrinti tvirtą informacijos slėpimą [Sad21]. Skirtingi esybių moduliai negali pasiekti failų, esančių kitų esybių paketų viduje, failai su vidiniais skaičiavimais laikomi privačiais, o visa komunikacija tarp modulių įgyvendinama viešomis, specialiai tam skirtomis sąsajomis, kurios agreguoja privačias klases ir eksponuoja funkcijas aprašančias bendravimo protokolą dalykinės srities funkcionalumui įgyvendinti. Dauguma programavimo kalbų, klasių bei funkcijų

privatumą gali užtikrinti pasiekiamumo modifikatoriais, į kuriuos atsižvelgia programavimo kalbos kompiliatorius.

```
/
├── pirkimai
│   ├── viešas PrekėsPirkimoServisas // eksponuoja sąsają pirkimui atlikti
│   ├── privatus MokesčiųSkaičiuotojas // skaičiuoja prekės mokesčius
│   └── privati PrekėsRepozitorija // atsakinga už darbą su duomenų baze
└── mokėjimai
    ├── viešas MokėjimųServisas // eksponuoja sąsają mokėjimui atlikti
    ├── privati MokėjimųRepozitorija // atsakinga už darbą su duomenų baze
    └── privatus ValiutosKonvertavimas // konvertuoja pirkimų valiutą
```

Šis metodas turi kelis trūkumus. Vienas jų - paketų struktūra neteikia informacijos apie bendrą sistemos architektūrą, pagrindinius programinius sluoksnius, kadangi visos klasės yra sudėtos į paketus pagal dalykinę sritį, o ne techninį sluoksnį. Taip paketų pavadinimų pasirinkimas užtrunka ilgiau, nes reikia išsirinkti tinkamus, atspindinčius tikslią produkto dalykinę sritį pavadinimus, neįvedant papildomos sumaišties.

# 3. Kodo skirstymo metodai realiose sistemose

## 3.1. Sistemų pasirinkimas

Norint įvertinti kodo skirstymo metodus egzistuojančiose programose, buvo pasirinktos Java programavimo kalba parašytos sistemos, kurių kodas viešai pasiekiamas GitHub repozitorijose. Sistemų aibė sudaryta iš populiarių, turinčių didelį lankytojų skaičių repozitorijų. Prioritetas buvo skirtas sistemoms, kurių dalykinė sritis reikalauja stabilumo bei patikimumo, pasižymi dideliu duomenų srautu.

Norint geriau suprasti programinio kodo grupavimo tendencijas, buvo pasirinktos skirtingo tipo programos – tiek į vartotojus orientuoti produktai, tiek sistemų infrastruktūros komponentai, duomenų bazės ar techninės bibliotekos, naudojamos kitų programų.

## 3.2. Sistemų analizės procesas

Sistemos analizuojamos rankiniu būdu, žiūrint į programinį kodą, naviguojant per sistemos direktorijų medžio struktūrą, įvertinami sutiktų paketų pavadimai, kodo kompleksiškumas, klasių tarpusavio komunikavimas, gerųjų praktikų laikymasis. Toks rankinis kodo analizės procesas yra ganėtinai subjektyvus, kadangi neįmanoma nuodugniai peržiūrėti viso programinio kodo, todėl, neįvertinus tam tikrų sistemos niuansų, galima priimti netikslias išvadas. Kad būtų sumažintas analizės subjektyvumas bei norint supaprastinti rankinius procesus, buvo sukurti keli įrankiai, pateikiantys struktūrizuotus ir mažiau žmogiškoms klaidoms jautrius analizės rezultatus:

### 3.2.1. Vaizduojami visi repozitorijos paketų pavadinimai

Kotlin programavimo kalba parašytas pagalbinis įrankis, išvedantis visus repozitorijos paketų pavadinimus (programėlės pavyzdys randamas priede numeris 1). Rezultatai yra naudojami vaizduojant dažniausių pavadinimų žodžių debesį angl. *world cloud*, kuris parodo visus paketų pavadinimus ir jų pasikartojimą. Tai naudinga norint geriau suprasti sistemą ir nustatyti paketų skirstymo metodą.

### 3.2.2. Vaizduojamas repozitorijos paketų medis

Kotlin programavimo kalba parašyta pagalbinė programėlė, kuri vaizduoja repozitorijos paketų medį Latex dirtree elementu, kuris naudojamas latex darbuose nubraižyti direktorijų struktūrą (programėlės pavyzdys randamas priede numeris 2). Gautas rezultatas leidžia įterpti gautą direktorijų medį į Latex įrankiu parašytus darbus. Tai padeda pamatyti bendrą sistemos architektūrą, sistemos kompleksiškumą, suprasti paketų gylį bei granuliarumą.

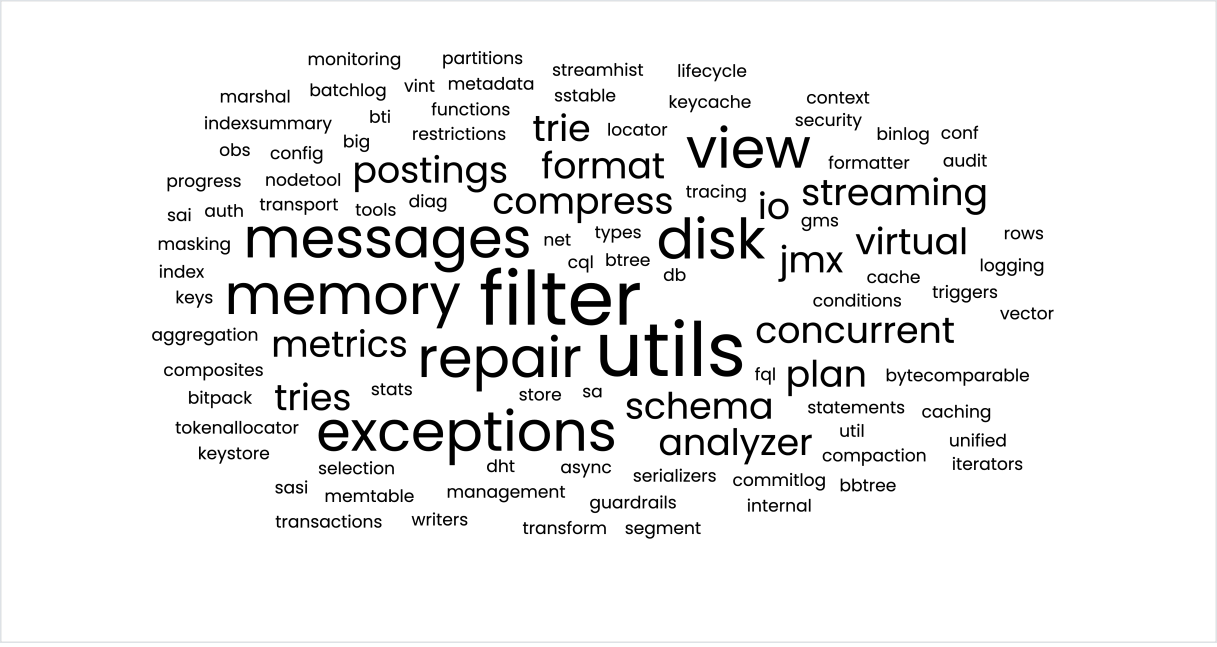
## 3.3. Apache Cassandra

### 3.3.1. Apache Cassandra sistemos aprašymas

Apache Cassandra yra Java programavimo kalba parašyta, lengvai besiplečianti (angl. *highly-scalable*) įrašų saugykla, naudojanti particijas duomenimis saugoti. Įrašų eilės yra suorganizuotos į lenteles su privalomu pirminiu raktu. Duomenų saugojimas particijomis reiškia, jog Cassandra savarankiškai išskaido duomenis per kelias fizines mašinas, užtikrinant duomenų saugumą ir greitesnį jų skaitymą. Kaip ir įprastos reliacinės duomenų bazės, Cassandra skirsto duomenis pagal eilutes ir stulpelius [Cas09]. Kadangi Apache Cassandra duomenų saugykla yra įrankis, naudojamas kitose sistemose, jis turi būti patikimas ir stabilus, naujos jo versijos neturi daryti neigiamos įtakos jį naudojančioms sistemoms. Paketų skirstymo metodas šioje sistemoje yra geras pavyzdys, kadangi šis būdas padeda įgyvendinti lengvai besiplečiančią bei patikimą sistemą.

### 3.3.2. Apache Cassandra programinio kodo analizė

Išnagrinėjus Apache Cassandra programinį kodą bei sugeneravus paketų pavadinimų debesies vizualizaciją, matomą paveikslėlyje žemiau, galima nustatyti, jog kodas nėra grupuojamas aiškiai apibrėžtu metodu. Dalis paketų, pavyzdžiui, index, transactions, notifications, hints - skirsto programinį kodą pagal dalykinės srities esybes - duomenų bazės ideksai, transakcijos. Tačiau programiniame kode taip pat yra direktorijų, kurios grupuoja kodą pagal jo techninius elementus - cache, net, config, security.




1 pav. Cassandra paketų pavadinimų debesų vizualizacija

Nors ši sistema neseka vieno klasių grupavimo metodo, tam, kad būtų užtikrintas geresnis kodo skaitomumas, keli, bendrus bruožus turintys paketai yra sugrupuoti po tėviniu paketu. Tai galima matyti sistemos paketų stuktūros fragmente:

```
/
├── locator
├── tools
│   └── nodetool
│       └── formatter
├── transport
│   └── messages
├── cql3
│   ├── statements
│   │   └── schema
│   ├── functions
│   │   ├── types
│   │   │   └── exceptions
│   │   └── masking
│   └── selection
├── cache
├── security
│   └── jmx
└── triggers
    └── tokenallocator
```

Toks paketų grupavimas keliais lygiais į subpaketus ir tėvinius paketus padeda geriau suprasti kontekstą, kuriame egzistuoja specifinės klasės, bei jų sąryšius.

### 3.3.3. Idealiai suskirstyto kodo kriterijų įgyvendinimas:

- Informacijos slėpimas (inkapsuliacija) - šis paketų skirstymo metodas, naudojant kelių lygių hierarchiją, nėra pilnai išnaudojamas. Sistemos kūrėjai neišnaudoja Java kalbos galimybių apriboti klasių pasiekiamumą su prieigos modifikatoriais - tėviniai paketai naudoja klases esančias sub paketuose, nėra aiškių gairių, kokios klasės gali būti pasiekiamos už paketo ribų. Toks sprendimas tikriausiai yra priimtas norint užtikrinti paprastą kodo kūrimo procesą, neapsunkinant darbo sudėtingomis architektūrinių sprendimų gairėmis, kurios nelaikomis būtinomis, jei esamoje sistemos būsenoje nesukelia rimtų problemų.
- Sistemos plėtros valdymas - kadangi paketų skirstymo metodas naudoja kelių lygiu hierarchiją, augant bendram failų skaičiui, failų skaičius vienam pakete nebutinai pasieks sunkiai naviguojamą lygi, taip leidžiant kontroliuoti sistemos plėtrą.
- Užuominos į sistemos architektūrą - paketų struktūra nėra tokia nuosekli jog teiktų papildomos informacijos apie sistemos architektūrą.
- Užuominos į verslo dalykinę sritį - paketų struktūra nėra tokia nuosekli jog teiktų papildomos informacijos apie sistemos verslo dalykinę sritį.
- Pagalba prisidedant prie sistemos kūrimo - paketų struktūra nėra tokia nuosekli jog prisidėtų prie sistemos kūrimo.

Įvertinus šią sistemą galima pastebėti, kad naudojant hibridinį skirstymo metodą, kur dalis paketų pavadinti pagal dalykinės srities esybes, o kiti pagal funckionalumą, dingsta nuoseklumas ir

prarandama galimybė suteikti informaciją tiek apie sistemos architektūrą, tiek apie verslo dalykinę sritį.

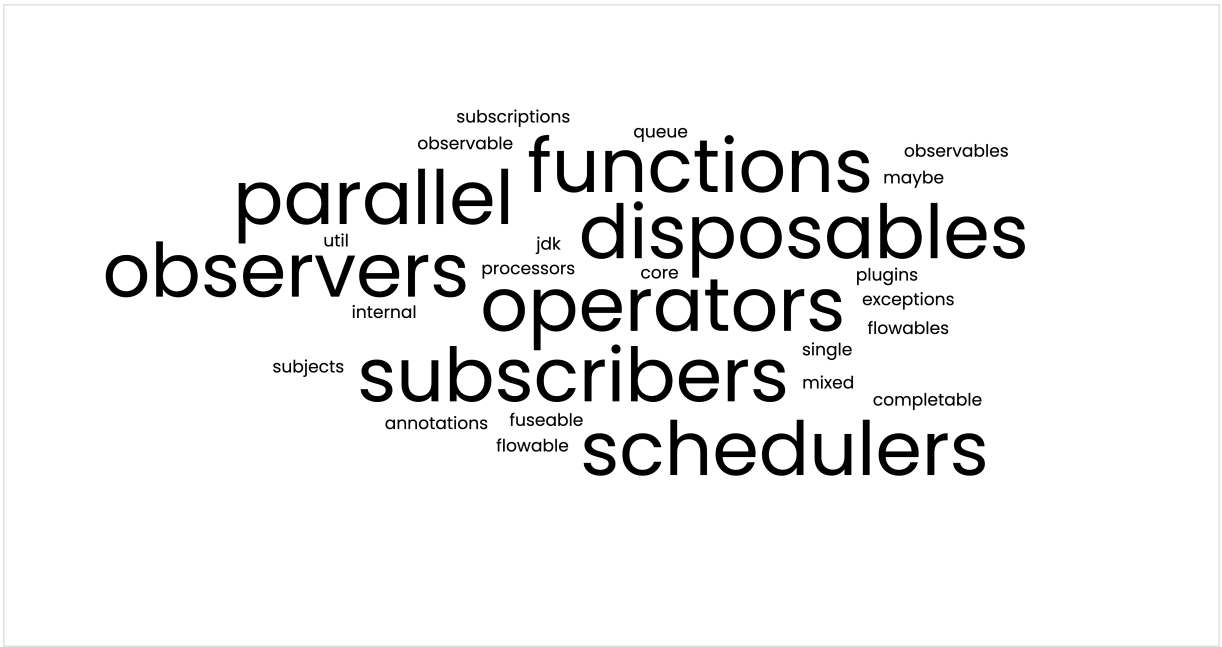
## 3.4. RxJava

### 3.4.1. RxJava sistemos aprašymas

RxJava yra Java programavimo kalba parašyta biblioteka, skirta rašyti sistemas, dirbančias su asinchroninėmis, įvykiais paremtomis duomenų sekomis. Ši biblioteka naudoja stebėtojo (angl. *observer*) modelį valdyti įvykių ir duomenų sekas. Taip pat, suteikia operatorius, leidžiančius deklaratyviai komponuoti duomenų manipuliacijas filtravimui, transformavimui, paslepiant žemo lygio problemas (lygiagretų programavimą, programinių gijų (angl. *thread*) valdymą bei sinchronizaciją [RxJ13]). Šis programinis kodas yra geras kandidatas analizei todėl, nes tai nėra standartinė sistema su aiškiomis dalykinės srities esybėmis, o techninė biblioteka, skirta naudojimui kitose sistemose.

### 3.4.2. RxJava programinio kodo analizė

Išnagrinėjus RxJava programinį kodą bei sugeneravus paketų pavadinimų debesų vizualizaciją, matomą paveikslėlyje žemiau, galima nustatyti, kad programinis kodas yra labai tvarkingas, paketams naudojami trumpi ir aiškūs pavadinimai, kodas suskirstytas į paketus pagal dalykinės srities esybes. Nors tarp paketų vardų yra tokių raktažodžių kaip functions, parallel, kurie iš pirmo žvilgsnio neatrodo kaip dalykinės srities esybių vardai, o labiau pavadinimai, nusakantys techninius sluoksnius, vis dėlto tai yra dalykinės srities esybės. Kadangi ši sistema yra ne produktas, naudojamas vartotojų, o biblioteka, skirta kitoms sistemoms, todėl ir jos dalykinės srities esybės apibūdina tokius komponentus, kurie kitose sistemos yra naudojami adresuojant funkcionalumo problemas.




2 pav. RxJava paketų pavadinimų debesų vizualizacija

Kadangi biblioteka yra sąlyginai nedidelė, beveik nėra kelių lygių paketų hierarchijos, visi pagrindiniai elementai yra pirmame direktorijos lygmenyje. Tai matome RxJavos paketų medyje:

```
/
├── parallel
├── core
├── operators
├── plugins
├── disposables
├── internal
│   ├── jdk8
│   ├── fuseable
│   ├── operators
│   │   ├── parallel
│   │   ├── mixed
│   │   ├── maybe
│   │   ├── single
│   │   ├── flowable
│   │   ├── observable
│   │   └── completable
│   ├── util
│   ├── disposables
│   ├── subscriptions
│   ├── queue
│   ├── functions
│   ├── subscribers
│   ├── observers
│   └── schedulers
├── observables
├── exceptions
├── annotations
├── processors
├── flowables
├── subjects
├── functions
├── subscribers
├── observers
└── schedulers
```

### 3.4.3. Idealiai suskirstyto kodo kriterijų įgyvendinimas:

- Informacijos slėpimas (inkapsuliacija) - ši sistema naudoja paketus užtikrinti klasių enkapsuliaciją - individualūs funkcionalumai yra talpinami skirtinguose paketuose, o visos klasės, kurių pasiekiamumas yra apribotas (pavyzdžiui, jos negali būti paveldėtos arba jų konstruktoriai yra privatūs) yra patalpintos internal pakete. Programuotojui pamačius paketą pavadinimu internal (vidinis), yra aišku, jog šios klasės nėra skirtos pasiekti už bibliotekos ribų.

- Sistemos plėtros valdymas – kadangi paketai suskirstyti pagal dalykinės srities esybes, augant esybių skaičiui, paketų skaičius gali labai išaugti ir naviguoti per sistemą pasidarytų nepatogu. Ateityje, norint sustabdyti šią problemą, reikėtų atsisakyti vieno lygio paketų medžio hierarchijos ir tam tikrų esybių paketus grupuoti po kitų, tėvinių esybių paketais.
- Užuominos į sistemos architektūrą – paketų struktūra neteikia papildomos informacijos apie sistemos architektūrą.
- Užuominos į verslo dalykinę sritį – šioje sistemoje paketų hierarchija teikia aiškias užuominas į verslo dalykinę sritį, kadangi kiekvienas paketo pavadinimas atspindi dalykinės srities esybę.
- Pagalba prisidedant prie sistemos kūrimo – kadangi visas kodas yra suskirstytas pagal dalykinės srities esybes, nauji žmonės, norintys pakeisti specifinį funkcionalumą, beveik visą kodą ras vienoje vietoje.

## 3.5. Analizės išvados

Įvertinus egzistuojančias sistemas yra matoma, jog yra teikiamas prioritetas paketų skirstymo metodui pagal dalykinės srities esybes. Jis sutinkamas net sistemose, kurios neturi griežtų programinio kodo skaidymo taisyklių. Taip pat matoma, kad iš paketų, pavadintų pagal dalykinės srities esybes, pavadinimų galima susidaryti aiškų vaizdą apie sistemos funkcionalumą. Dar viena tendencija, matoma tarp skirtingų sistemų – sistemose nėra apibrėžtų griežtų gairių arba niuansų, kaip naudoti paketus, užtikrinant geresnę kodo kokybę, pavyzdžiui, palaikant griežtą subpaketų hierarchiją, apribojant priklausomybes tarp tėvinių paketų ir subpaketų, apibrėžiant dedikuotas klases, kurios vienintelės būtų pasiekiamos komunikacijai už paketų ribų. Cassandra sistemos analizėje galima pastebėti, jog hibridinis, aiškiai neapibrėžtas metodas nesuteikia pilnos informacijos nei apie dalykinės srities esybes, nei apie sistemos sluoksnius, o tik prideda painumo. Į kodo skirstymą žiūrima ganėtinai pragmatiškai, neapibrėžiant procesų, teikiant prioritetą programinės įrangos kūrimo greičiui ir paprastumui vietoje suvaržymų, suteikiančių kodo stabilumą.

# 4. Gairės kodo skirstymui

Surinkus ir įvertinus teorinę informaciją apie galimus kodo skirstymo metodus, jų privalumus ir trūkumus, bei išanalizavus kodo skirstymo paketais pavyzdžius, sutinkamus realiose sistemose, galima aprašyti gaires, kaip reikėtų skirstyti programinį kodą. Idealaus metodo, kaip skirstyti programinį kodą nėra, kiekvienas pasirinkimas turi savo privalumų bei trūkumų, vieni metodai gali turėti teorinių privalumų, bet jų įgyvendinimas yra pernelyg brangus, kiti metodai gali atrodyti labai perspektyvūs tačiau panaudojami tik labai specifinėse sistemose. Į šį faktą yra atsižvelgiama apibrėžiant gaires - nors gairių rezultatas yra imtis taisyklių, kurių laikantis galima prisidėti pasiekiant patikimą, lengvai plečiamą ir prižiūrimą sistemą, yra suprantama, kad tam tikrais atvejais, specifinės gairės gali būti ne visai tinkamos. Programuotojai gali sąmoningai jų nesilaikyti, įvertinę gairės privalumus ir trūkumus. Tam, kad būtų galima objektyviau įvertinti kraštutinius atvejus, kai gairė sukelia daugiau problemų nei privalumų, prie kiekvienos gairės yra pateiktas jos ribojimų bei trūkumų sąrašas.

## 4.1. Laikytis vientisumo

Gali pasirodyti, jog skirstyti klases į paketus pagal dalykinės srities funkcionalumą yra pranašesnis sprendimas - augant sistemai, vykdant pakeitimus specifiniuose funkcionalumuose, programuotojui nereikės ieškoti kodo per kelis skirtingus paketus, taip pat toks metodas suteikia naudingos informacijos apie dalykinės srities esybes esančias sistemoje (tai galima matyti RxJava sistemos analizės metu, kai iš paketų, pavadintų pagal dalykinės srities esybes, pavadinimų galima susidaryti vaizdą apie sistemos funkcionalumą). Deja, ne visada galima grupuoti kodą pagal ši metodą:

- Komanda, sudaryta iš žmonių, kuriems šis metodas yra painus, nes jie anksčiau yra dirbę su sistemomis, kuriose kodas sugrupuotas pagal techninius sluoksnius.
- Techninės sistemos dalykinės srities esybės gali būti panašios į techninių sluoksnių, todėl sunku parinkti neklaidinančius paketų pavadinimus
- Naudojamos bibliotekos ar karkasai reikalauja skirstyti kodą pagal sistemos sluoksnius. Pavyzdžiui, *Java Spring* karkase, kad būtų galima gauti karkaso įgyvendintas funkcijas darbui su duomenų bazėmis, aprašomos sąsajos (angl. *interfaces*) aktualioms esybėms, įgyvendinančios Repository sąsają (angl. *interface*). Jos patalpinamos į tą patį paketą ir *@EnableJpaRepositories("kelias.iki.repozitorijų")* anotacijos pagalba, jų vieta užregistruojama *Spring* karkase. Dėl šio veiksmo programos veikimo metu karkasas pateiks aprašytų sąsajų igyvendinimus, ten kur jų reikia. Nors ir įmanoma nurodyti kelias skirtingas repozitorijas, atsiranda papildomos konfigūracijos, apsunkinančios kūrimą.

Todėl yra svarbiau užtikrinti kodo skirstymo vientisumą - geriau visas klases sugrupuoti pagal sistemos techninius sluoksnius, o ne naudoti hibridinį variantą, dalį paketų pavadinant pagal sluoksnius, o kitus pagal dalykinę sritį. Cassandra sistemos analizėje galima pastebėti, jog hibridinis metodas nesuteikia pilnos informacijos nei apie dalykinės srities esybes, nei apie sistemos sluoksnius, o tik prideda painumo. Dalies šių problemų būtų galima išvengti, skirstant kodą vien tik pagal

techninius sluoksnius. Taip pat yra labai svarbu sistemos dokumentacijoje pateikti informaciją, kaip yra grupuojamas programinis kodas ir kokių gairių reikia laikytis konkrečioje sistemoje, taip užtikrinant, kad naujai prisijungę žmonės turės informaciją, kaip dirbti su programiniu kodu, nepakenkiant sistemos vientisumui.

## 4.2. Naudoti kelių lygių hierarchiją

Dideli paketai turėtų būti išskaidyti į mažesnius paketus, sugrupuojant kelias bendrus bruožus turinčas dalykinės srities esybes ar techninius sluoksnius į paketus, esančius kito, tėvinio paketo viduje.

Skirstant kodą pagal dalykinės srities esybes, paketų medis galėtų atrodyti taip:

```
/
└── užduotys
    ├── įgaliotiniai
    └── perspėjimai
```

Jei reikalinga skirstyti kodą į paketus pagal techninius sluoksnius, paketų medis galėtų atrodyti taip:

```
/
├── servisai
├── esybės
│   ├── duombazės
│   └── http
└── klientai
    ├── duombazės
    └── http
```

Vadovaujantis šia gaire, vietoj kelių vieno lygio paketų su daugybe klasių, turėsime daug smulkių paketų, išsidėsčiusių keliais lygiais. Tokia paketų medžio hierarchija prisideda prie sistemos plėtros valdymo kriterijaus, kadangi visas kodas susijęs su tam tikru komponentu yra vienoje vietoje, tačiau nėra milžiniškų direktorijų su daugybe failų vienoje vietoje[Sad21]. Maži paketai yra lengviau suprantami, todėl laikantis šios gairės sistema suteikia papildomą pagalbą naujiems žmonėms, prisidedantiems prie jos kūrimo. Ši gairė nėra labai naudinga paprastose sistemose, kur vieno funkcionalumo kodas telpa į kelis failus, ir paketai nėra perpildyti, todėl yra priimtina ją pridėti sistemai pasiekus tam tikrą dydį ar kompleksiškumą.

## 4.3. Vaikiniai subpaketai neturėtų pristatyti naujų konceptų

Pagrindinė kodo logika turėtų būti laikoma tėviniame pakete, paslėpta abstrakcijomis (angl. *interface*). Subpaketai turėtų įgyvendinti minėtas abstrakcijas, tačiau neaprašyti naujų funkcionalumų, naujo bendradarbiavimo tarp skirtingų modulių. Galima būtų teigti, jog subpaketai neturėtų pridėti papildomo funkcionalumo, tik papildomas detales. Jei vaikiniai paketai pristato naujus

konceptus, funkcionalumus, programuotojai negali susidaryti bendro vaizdo apie funkcionalumą, neperžiūrėję visų subpaketų[jav17]. Laikantis šios gairės, ši problema yra išvengiama ir naudojamas kodo grupavimo metodas tenkina idealiai suskirstyto kodo kriterijus teikti užuominas į sistemos architektūrą ir į verslo dalykinę sritį, kadangi visa ši informacija gali būti rasta tėviniuose paketuose. Ši gairė iš dalies apriboja anksčiau minėtą gairę (skirstyti paketus keliais lygiais), kadangi ne visi lygiai yra lygiaverčiai ir dalis funcionalumo turi likti tėviniame pakete.

## 4.4. Tėviniai paketai turėtų nepriklausyti nuo vaikinių paketų

Sistemos, kurių paketų moduliai priklauso vienas nuo kito abiem kryptimis – tėviniai nuo vaikinių ir vaikiniai nuo tėvinių, sukuriant žiedines priklausomybes (angl. *circular dependencies*) pasižymi labai sudėtinga priežiūra. Pakeitimai retai kada buna lokalizuoti ir paveikia kelias klases. Taip pat tokia sistema praleidžia galimybę komunikuoti su sistemos skaitytojų nurodant aiškią sistemos architektūrą ir linijinę komunikaciją tarp skirtingų klasių[jav17]. Tam, kad būtų išvengta šių problemų, tėviniai paketai turėtų naudoti tik abstrackijas (angl. *interface*), aprašytas pagal ankstesnę gairę, komunikacijai su vaikiniais paketais, apibrėžiant aiškia, vienpusę priklausomybę tarp skirtingų paketų. Ši gairė supaprastina sistemos priežiūrą – lokalizuoja pakeitimus į mažus modulius, bei leidžia aiškiau matyti, kaip klasės bendrauja tarpusavyje vykdant specifinius dalykinės srities funkcionalumus. Ji taip pat užtikrina stiprią enkapsuliaciją – yra apibrėžiamas tvirtas, abstrakcija paremtas atskyrimas tarp tėvinių ir vaikinių paketų, paslepiantis sąsajų įgyvendinimo detales. Ši gairė turi vieną svarbų trūkumą – ją gan sunku palaikyti, kiekvieno pakeitimo metu programuotojai turi pastebėti, ar neatsirado priklausomybė nuo vaikinio paketo.

## 4.5. Naudoti dedikuotas klases bendravimui tarp paketų

Ne visos paketo viduje esančios klasės turėtų būti pasiekiamos iš išorės. Bendravimui tarp paketų turėtų būti sukurtos dedikuotos aplikacijų programavimo sąsajos (API) klasės, užtikrinant, kad kiekvienas paketas yra atskiras, uždaras modulis. Tai suteikia informacijos slėpimą (enkapsuliaciją), kadangi pakeitimai modulio viduje nedaro įtakos kitiems paketams, kol nėra pakeistos dedikuotos aplikacijų programavimo sąsajos klasės.

Architektūra naudojant programavimo sąsajos klases galetų atrodyti taip:

```
/
├── prekė
│   ├── PrekėsAPI // pasiekiama už modulio ribų, funkcionalumas reikalingas
│   │   kitiems moduliams
│   └── … // daugiau klasių modulyje, nepasiekiamų už modulio ribų
├── pirkimai
│   ├── PirkimųAPI // pasiekiama už modulio ribų, funkcionalumas reikalingas
│   │   kitiems moduliams
│   ├── PirkimųRepozitorija // pasiekia prekės logiką per PrekėsAPI, klasė
│   │   nepasiekiama už modulio ribų
│   └── … // daugiau klasių modulyje, nepasiekiamų už modulio ribų
└── detalės
    ├── DetaliųAPI // pasiekiama už modulio ribų, funkcionalumas reikalingas
    │   kitiems moduliams
    ├── DetaliųRepozitorija // pasiekia prekės logiką per PrekėsAPI, klasė
    │   nepasiekiama už modulio ribų
    └── … // daugiau klasių modulyje, nepasiekiamų už modulio ribų
```

API klasės turetų būti aprašomos tik tėviniuose paketuose, tam jog neprieštarautų gairei jog vaikiniai paketai neturėtų pristatyti naujų konceptų. Naudojant sąsajas (angl. *interface*) API klasėmis aprašyti, užtikrinama, kad jų funkcionalumas gali būti naudojamas tiek kituose tėviniuose moduliuose, tiek jų vaikiniuose paketuose, nesukuriant priklausomybių nuo vaikinių paketų (Klasės, įgyvendinančios aprašytą sąsają priklauso nuo API kontrakto, o ne nuo API naudotojo)

## 4.6. Neturėti pasikartojančių raktažodžių paketo ir klasės pavadinime

Žodžiai klasės pavadinime neturėtų kartoti žodžių esančių pakete / direktorije. Pavyzdžiui vietoje *vartotojas/VartotojoServisas* geriau naudoti *vartotojas/Servisas*. Tokiu atveju pervadinant paketą, nereikia pervadinti jame esančių klasių[Sad21], dėl to yra lengviau prižiūrėti sistemą, nes pakeitimai paketų pavadinimuose yra mažesni, lengviau suprantami. Tai taip pat pašalina nereikalingą pasikartojimą paketu hierarchijoje. Modernios integruotos kūrimo aplinkos sugeba ieškoti klasių, atsižvelgiant į visą failo kelią, todėl supaprastinti vardai neapsunkina paieškos - net esant kelioms klasėms pavadinimu *servisas*, ieškant *vartotojo/servisas* bus rasta butent ta klasė, esanti vartotojo pakete. Tačiau dirbant su labai paprastomis kūrimo aplinkomis, naudojant šią konvenciją be pasikartojančių raktažodžių, klasių paieška tampa komplikuota. Pavyzdžiui, paieška *vartotojo/servisas* rezultatų neras, o paieška *servisas* grąžins visas klases pavadinimu *servisas*. Taip pat, kai kurie priklausomybių pateikimo (angl. *dependency injection*) karkasai pagal nutylėjimą parenka reikiamas priklausomybes naudojant klasių pavadinimus, todėl, sekant šią gairę, karkasas neveiks be papildomų kvalifikatorių.

# 5. Gairių pritaikymas

Kad būtų galima geriau suprasti aprašytų gairių panaudojimą, jas pritaikysime egzistuojančiai sistemai. Tam buvo pasirinkta programėlė *PocketDiary*, skirta pateikti kasdienės veiklos ir susijusių jausmų apžvalgą [Sab21]. Ši sistema pasirinkta todėl, nes ji yra atviro kodo, parašyta Java programavimo kalba, sąlyginai paprasta, bei nesilaiko jokių apibrėžtų gairių.

Prieš pritaikant aprašytas gairės, sistemos struktūra atrodo taip:

```
/
├── models
├── adapters
├── activities
│   ├── tutorial
│   ├── entry
│   └── welcome
└── daos
```

Galima pastebėti, jog klasės nėra sugrupuotos į paketus vienu metodu - *models, adapters, daos, activities* yra techniniai sluoksniai, o *entry, welcome, tutorial* yra dalykinės srities esybės. Tam, jog butų užtikrintas vientisumas, pasirenkama skirstyti paketus pagal dalykinės srities esybes - yra išrenkamos visos sistemoje esančios dalykinės srities esybės, pagal jas sukuriami paketai ir juose sugrupuojamos sistemos klasės:

```
/
├── achievements
├── mood
├── actions
├── tutorial
├── entry
├── welcome
├── home
└── notifications
```

Po šių pakeitimų sistemos paketų skirstymo metodas tampa vientisas. Paketai išskirstyti keliais lygiais, ten, kur reikia vaikines esybes sugrupuojant į tevinės esybės paketus.

Norint padaryti sistemą lengviau skaitomą, visus dalykinės srities konceptus aprašome tėviniuose paketuose kaip sąsajas.

Kad tai būtų pasiekta, įvertiname visus sistemoje rastus dalykinės srities funkcionalumus ir aprašome juos apibendrinančias sąsajas:

```
entry
├── Entry
├── EntryDao
├── EntryWithActions
├── EntryWithActionsDao
└── action
```

Šios sąsajos taip pat užtikrina, jog tėviniai paketai nepriklauso nuo vaikinių paketų – visa komunikacija vyksta abstrakčiomis sąsajomis, nesukuriant tiesioginės priklausomybės.

Siekiant padidinti informacijos slėpimą (enkapsuliaciją), naudojant Java programavimo kalbos pasiekiamumo modifikatorius, sumažinamas sąsajų, kurios neturėtų būti viešos, pasiekiamumas, jog jos būtų nepasiekiamos už paketų ribų. Aprašomos naujos sąsajos dedikuotos viešai, tarppaketinei komunikacijai, yra apibrėžiamos dviejų tipų viešai prieinamos sąsajos su *API* ir *Activity* raktažodžiais Sąsajos su *API* raktažodžiu yra skirtos tiekti funkcionalumui kitoms esybėmis, o sąsajos su *Activity* raktžodžiu skirtos Android SDK karkasui:

```
/
├── achievements
│   └── AchievementsActivity
├── mood
├── actions
├── tutorial
│   └── TutorialActivity
├── entry
│   └── EntriesAdapterAPI
│       ├── NewEntryActivity
│       ├── NewEntryNoteActivity
│       └── ViewEntryActivity
├── welcome
│   ├── WelcomeAActivity
│   ├── WelcomeBActivity
│   ├── WelcomeCActivity
│   └── WelcomeDActivity
├── home
│   ├── MainActivity
│   ├── HomeScreenActivity
│   └── RoomDatabaseAPI
└── notifications
    └── ReminderBroadcastAPI
```

Galima pastebėti, jog paketų pavadinimai ir sąsajos turi daug pasikartojančių raktažodžių, kuriuos galima būtų supaprastinti, sumažinant perteklinę informaciją:

```
/
├── achievements
│   └── Activity
├── mood
├── actions
├── tutorial
│   └── Activity
├── entry
│   └── AdapterAPI
│       ├── NewActivity
│       ├── NewNoteActivity
│       └── ViewActivity
├── welcome
│   ├── AActivity
│   ├── BActivity
│   ├── CActivity
│   └── DActivity
├── home
│   ├── Activity
│   ├── HomeScreenActivity
│   └── RoomDatabaseAPI
└── notifications
    └── ReminderBroadcastAPI
```

Po pakeitimų kodo paketų struktūroje, *PocketDiary* sistema laikosi visų aprašytų kodo skirstymo gairių. Pradinis sistemos kodas pasiekiamas repozitorijoje PocketDiary[1], o paketų struktūra po gairių pritaikymo pasiekiama repozitorijoje KursinisPraktika[2]

[1] https://github.com/Sabo2k/pocket-diary

[2] https://github.com/MartynaUb/kursinis-praktika

# Rezultatai

1. Darbo metu buvo įvertinti reikalavimai, kada sistema gali būti laikoma teisingai įgyvendinta. Atsižvelgiant į įvertintus reikalavimus aprašyti kriterijai kaip kodo skirstymo metodas galėtų prisidėti prie tinkamesnio sistemos įgyvendinimo.
2. Naudojant gautus kriterijus buvo įvertinti skirtingi teoriniai būdai programinio kodo skirstymui, bei išanalizuotos tikros, industrijoje egzistuojančios kompiuterinės sistemos, tam jog būtų galima suprasti, kaip nagrinėti teoriniai kodo skirstymo metodai naudojami praktikoje, bei pamatyti sistemų spragas, atsiradusias dėl netinkamo kodo skirstymo.
3. Aprašytos gairės, kaip reiketų skirstyti programinį kodą, jog jis atitiktų nustatytus kriterijus, nusakančius kaip skirstymo metodas turėtų padėti siekti geriau įgyvendintos sistemos.
4. Sukurta programėlė, išvedanti paketų pavadinimų sąrašą.
5. Pakoregavus atviro kodo repozitorijos skirstymą parodoma, kad gairės gali būti suderinamos.

# Išvados

1. Svarbu, kad sistemos kodas būtų paskirstytas nuosekliu, visiems komandos nariams suprantamu ir dokumentacijoje aprašytu būdu.
2. Atviro kodo sistemose, sutinkamose industrijoje, trūksta nuoseklumo programinio kodo skirstyme. Teisingai sugrupuoto kodo svarba nėra akcentuojama ir šioje srityje yra kur tobulėti.
3. Nors iš visų analizuotų sistemų nebuvo nė vienos, kuri laikytųsi visų aprašytų gairių, visos gairės bent dalinai buvo įgyvendintos skirtingose sistemose, suteikiant pasitikėjimo aprašytų gairių patikimumui.
4. Sukurtos kodo skirstymo į paketus gairės gali būti suderinamos ir naudojamos praktikoje.

# Šaltiniai


[Cas09] A. Cassandra. *Apache Cassandra*. 2009. [žiūrėta 2023-11-10]. Prieiga per internetą: https://github.com/apache/cassandra.

[jav17] javadevguy. Happy Packaging! 2017 [žiūrėta 2023-11-01]. Prieiga per internetą: https://javadevguy.wordpress.com/2017/12/18/happy-packaging/.

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

[RxJ13] RxJava. *RxJava*. 2013. [žiūrėta 2023-11-10]. Prieiga per internetą: https://github.com/ReactiveX/RxJava.

[Sab21] Sabo2k. *PocketDiary*. 2021. [žiūrėta 2024-01-10]. Prieiga per internetą: https://github.com/Sabo2k/pocket-diary.

[Sad21] M. Sadowski. Divide Your Codebase by Domains and Features To Keep It Scalable. 2021 [žiūrėta 2023-11-02]. Prieiga per internetą: https://betterprogramming.pub/divide-code-by-domains-and-features-and-keep-it-scalable-bb5bd66cf3d2.

[Sou] Sourcemaking. Lava Flow. [Sine anno] [žiūrėta 2023-11-02]. Prieiga per internetą: https://sourcemaking.com/antipatterns/lava-flow.

# Priedai

## Priedas nr. 1

## Programėlė repozitorijos paketų pavadinimams išvesti

```
val file = File("kelias iki analizuojamos repozitorijos")
// Iteruojama per visus repozitorijos failus / direktorijas
file.walk().forEach {
// Jei failas yra direktorija, išvedamas direktorijos pavadinimas
    if (it.isDirectory) {
        print(it.name + " ")
    }
}
```

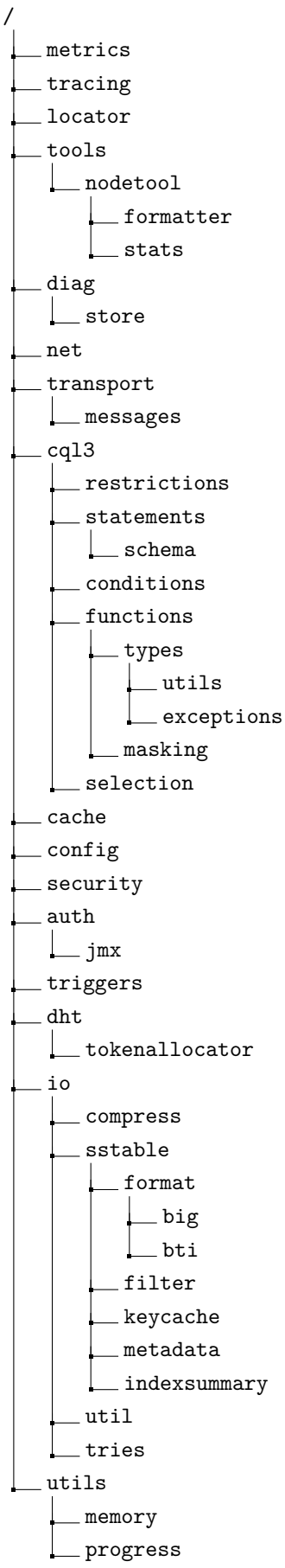
## Priedas nr. 2

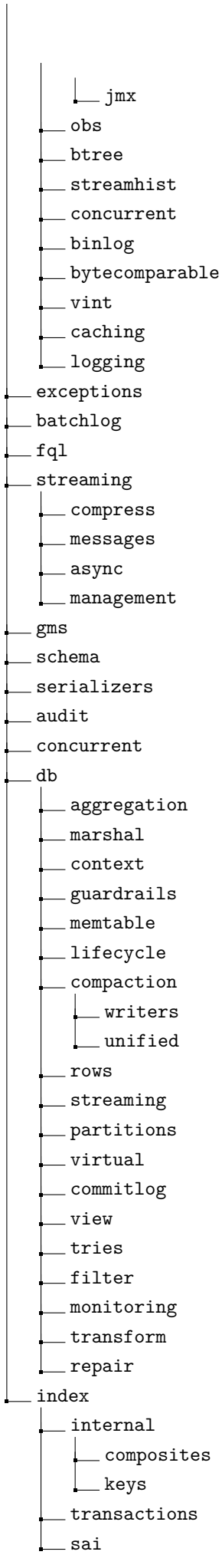
## Programėlė repozitorijos paketų medžiui išvesti

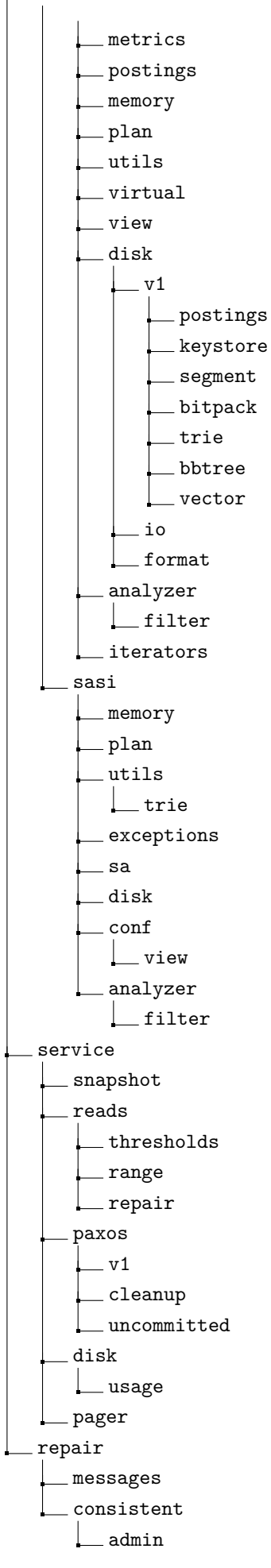
```
println("\\dirtree{%")
println(".1 {/} .")
// Iteruojama per visus repozitorijos failus / direktorijas
file.walk().forEach {
    // paskaičiuojamas reliatyvus failo kelias nuo pagrindinės direktorijos
(reikalinga paskaičiuoti direktorijos gyliui).
    val path = it.relativeTo(file).path.toString().split("/")
    if (path.last() != "" && it.isDirectory) {
        // išvedamas dirtree latex elementas su direktorijos pavadinimu ir nurodytų gyliu
        println(".${path.size + 1} {${path.last().replace("_", "\\_")}}.")
    }
}
println("}")
```

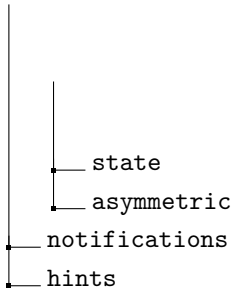
## Priedas nr. 3

## Pilnas Cassandra paketų medis




jmx
obs
btree
streamhist
concurrent
binlog
bytecomparable
vint
caching
logging
exceptions
batchlog
fql
streaming
compress
messages
async
management
gms
schema
serializers
audit
concurrent
db
aggregation
marshal
context
guardrails
memtable
lifecycle
compaction
writers
unified
rows
streaming
partitions
virtual
commitlog
view
tries
filter
monitoring
transform
repair
index
internal
composites
keys
transactions
sai


metrics
postings
memory
plan
utils
virtual
view
disk
v1
postings
keystore
segment
bitpack
trie
bbtree
vector
io
format
analyzer
filter
iterators
sasi
memory
plan
utils
trie
exceptions
sa
disk
conf
view
analyzer
filter
service
snapshot
reads
thresholds
range
repair
paxos
v1
cleanup
uncommitted
disk
usage
pager
repair
messages
consistent
admin


state
asymmetric
notifications
hints

## Priedas nr. 4

## Pilnas RxJava paketų medis